Adaptations dynamiques au contexte en informatique ambiante : propriétés logiques et temporelles

Nicolas Ferry

UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS

ÉCOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

T H È S E

pour obtenir le titre de

Docteur en Sciences

de l'Université de Nice - Sophia Antipolis
Mention : INFORMATIQUE

Présentée et soutenue par

Nicolas FERRY

Adaptations dynamiques au contexte en informatique ambiante : propriétés logiques et temporelles

Thèse dirigée par Michel RIVEILL

préparée au sein du laboratoire I3S, Equipe RAINBOW

soutenue le 07 décembre 2011

Jury :

Rapporteurs :	Yves DENNEULIN	-	Professeur, INPG Grenoble
	Philippe ROOSE	-	Docteur H.D.R., IUT de Bayonne
Directeur :	Michel RIVEILL	-	Professeur, Université de Nice
Co-Directeur :	Stéphane LAVIROTTE	-	Docteur, Université de Nice
Président :	Pascal ESTRAILLER	-	Professeur, Université de La Rochelle
Examinateur :	Jean-Michel BRUEL	-	Professeur, Université de Toulouse
Invité :	Jean-Yves TIGLI	-	Docteur, Université de Nice
Invité :	Eric PASCUAL	-	Ingénieur, CSTB

Remerciements

Mes remerciements s'adressent en premier lieu à Stéphane Lavirotte qui m'a encadré durant cette thèse. Je te remercie pour tes conseils, ta patience, tes relectures (en particulier, je te remercie de ne pas avoir compté le nombre de "s" et de "," que tu auras ajouté à mes écrits ...). Merci pour avoir toujours été présent, ton aide et tes encouragements. J'ai pris beaucoup de plaisir à travailler avec toi durant ces 3 années. Je remercie également Michel Riveill d'avoir suivi et dirigé cette thèse. Merci d'avoir été présent quand j'avais besoin de conseils et d'avoir toujours su trouver la "question qui tue". Je remercie également tout particulièrement Jean-Yves Tigli et Gaëtan Rey, pour m'avoir suivi, encouragé, relu et aidé tout au long de cette thèse, et ce, toujours dans une ambiance formidable. Gaëtan, merci de m'avoir aidé à finir cette thèse avec plus de sérénité et de m'avoir intégré à l'IUT.

Je remercie Philippe Roose, Yves Denneulin d'avoir accepté d'être les rapporteurs de cette thèse et pour leurs retours. Je les remercie également, ainsi que Jean-Michel Bruel et Pascal Estrailler de m'avoir fait l'honneur de participer à ma soutenance de thèse.

Je remercie également l'ensemble de l'"Ubiquarium team" pour l'ambiance incroyable qu'il y a eu tout au long de ces années. En particulier, je remercie "vinvin" et "simone". Vinvin (aka vinouze), merci pour le nombre de "meilleur film du monde" que tu as trouvé (même si je persiste à dire que :"non, transformers n'est pas le meilleur film du monde ...") et ton initiation à "ultra vomit". Simone, merci pour l'ambiance que tu as apporté et les discussions que l'on a eu. Mélaine, merci pour l'aide apportée sur le tisseur et ces évals. Sana merci d'avoir toujours eu le sourire. Enfin, "Mimi", merci pour les rigolades que l'on a eu et d'avoir su garder Kali comme un chef !

Durant ces trois années, j'ai fait parti de l'équipe Rainbow (ancienne et nouvelle version), j'en remercie tous les membres pour la superbe ambiance qui y règne. J'aimerai également remercier Sébastien et Mireille pour leurs conseils et leur aide durant le projet Faros. Ma thèse a également été réalisée en collaboration avec le CSTB. Aussi, je remercie Eric Pascual pour l'aide qu'il m'a apporté et Daniel Cheung pour les discussions intéressantes que nous avons eues et le terreau fertile apporté par sa thèse qui a donné naissance à la mienne.

Parallèlement à mes travaux de recherche, j'ai eu l'occasion d'enseigner en tant que moniteur. Je remercie Marc Gaëtano d'avoir bien voulu être mon tuteur et surtout de m'avoir montré comment enseigner. J'ai beaucoup apprécié les libertés données dans la rédaction de TP et examens que tu m'as laissé... tout cela dans une ambiance magique. Je tiens d'ailleurs ici à préciser que je t'ai "tué" plus souvent que tu ne m'as "tué". Je remercie pareillement les différentes "dream team d'env info" auxquelles j'ai pu participer. Christian, merci pour toute l'aide apportée dans la gestion de l'enseignement d'IaI. Je remercie également Jean-Yves et Stéphane de m'avoir fait confiance et proposé de faire des enseignements d'applications réparties et d'IAm jusqu'au cours en amphithéatre.

Enfin, je tiens à remercier ma famille. Tout d'abord mes parents, merci de m'avoir toujours laissé libre et soutenu dans mes choix, même dans les moments difficiles. Merci d'avoir fait de moi ce que je suis aujourd'hui. Mon Hélène, merci pour ta présence passée et à venir, tes conseils, ton amour et de m'avoir supporté malgré les grandes soirées d'écriture, les coups de blues ou ma mauvaise humeur. Ta présence m'a toujours permi de repartir du bon pied.

Résumé

En informatique ambiante, les applications sont construites en faisant interagir entre eux des objets informatisés et communicants appelés dispositifs. Parce que ces dispositifs peuvent être mobiles ou subir des pannes, cette infrastructure évolue dynamiquement et de manière imprévisible. Aussi, pour s'insérer de manière transparente dans leur environnement et fournir les fonctionnalités attendues par les utilisateurs, bien souvent plus pérennes que l'environnement sur lequel elles reposent, les applications doivent s'adapter dynamiquement à ces évolutions. Ces phénomènes variables poursuivant leur propre dynamique, le défi proposé aux mécanismes d'adaptation est d'être capable de les prendre en compte, avec une dynamique adaptée à chacun d'entre eux.

Dans cette optique, nous proposons un modèle architectural ainsi qu'un mécanisme d'adaptation. Le modèle architectural repose sur quatre niveaux organisés hiérarchiquement en fonction de leur complexité et de la dynamique qu'ils peuvent offrir. Nous lui associons un mécanisme d'adaptation qui, à partir du principe de séparation des préoccupations permet d'exprimer la variabilité du système. En raison de l'imprévisibilité de l'environnement, les ensembles d'adaptations qui seront déployées par les niveaux supérieurs de l'architecture ne peuvent pas nécessairement être anticipés à la conception. Aussi, grâce à un ensemble de propriétés logiques et temporelles, ces adaptations peuvent être composées de manière non-anticipée dans des temps de réponse adaptés. Le mécanisme d'adaptation proposé, appelé cascade d'aspects, est expérimenté en se basant sur les Aspects d'Assemblages et la plateforme d'exécution WComp.

Abstract

In ubiquitous computing, applications are built as a collaboration of computerized and communicating objects called devices. Because these devices can be mobile or subject to failures, this infrastructure evolves dynamically and unpredictably. Thus, to fit seamlessly into their environment and to provide the functionalities expected by users which are often more sustainable than the environment, applications must dynamically adapt to these changes. Each of these variable phenomena pursues its own dynamic. The challenge offered to adaptation mechanisms is to be able to consider them, with suitable dynamics.

For this purpose, we propose an architectural model and an adaptation mechanism. The architectural model is based on four levels organized hierarchically according to their complexity and to the dynamics they can offer. We combine to this architectural model an adaptation mechanism. Based on the separation of concerns principle, our mechanism allows us to consider the variability of the system. Due to the unpredictability of the environment, the sets of adaptations that will be deployed by the upper levels of the architecture may not have been anticipated at design time. Also, thanks to some logical and temporal properties, these adaptations can be composed in non-anticipated way and with appropriate response time. The proposed mechanism, called cascaded aspects, is implemented using Aspects of Assembly and the WComp execution platform.

Table des matières

Première partie

Introduction et analyse de l'état de l'art

CHAPITRE 1

Introduction

« Croire ou ne pas croire, cela n'a aucune importance. Ce qui est intéressant, c'est de se poser de plus en plus de questions »

Bernard Werber.

Sommaire

CE CHAPITRE constitue une introduction aux domaines de recherche évoqués dans cette thèse. Après avoir présenté notre cadre de travail qu'est l'informatique ambiante, nous étudierons les challenges auxquels nous nous intéresserons dans ce manuscrit. A partir de cet aperçu, nous soulèverons un ensemble de questions et les objectifs de cette thèse.

1.1 Introduction

Il y a vingt ans déjà, Mark Weiser [52] introduisait sa vision d'une informatique du futur. Il parlait alors d'une informatique invisible, présente en tout lieu, en toute chose. Aujourd'hui, avec la miniaturisation des matériels informatiques, de nombreux objets informatisés se trouvent disséminés dans notre quotidien et font peu à peu disparaître l'idée de l'ordinateur personnel comme le seul objet informatisé, ou encore de l'ordinateur comme l'assistant numérique universel. On parle alors d'intelligence ambiante. Désormais, cette intelligence ambiante est envisageable grâce aux évolutions apparues ces dernières années dans les domaines de la microélectronique, des télécommunications et de l'informatique [47]. Les avancés continuelles réalisées dans le domaine de la microélectronique permettent de construire des objets, dotés de capacités de calcul et de communication, toujours plus puissants et autonomes. Parallèlement à cela, les progrès réalisés dans les télécommunications, comme la démocratisation des communications sans fils, facilite les interconnexions entre ces objets. L'informatique, bénéficie de ces évolutions et a pour objectif de gérer ces objets et leurs interconnexions afin de rendre des services de manière transparente. Cette informatique, que l'on nomme « l'informatique ambiante », sera le cadre de travail des travaux de cette thèse.

1.2 Les défis de l'informatique ambiante

Un système ambiant se trouve dans un environnement et repose sur un ensemble d'entités qui peuvent interagir les unes avec les autres. Une entité peut être un « êtres vivant » comme un utilisateur par exemple, ou un système informatisé. Ces systèmes reposent sur des objets informatisés **hétérogènes** [46, 44] qui ont une existence physique, par exemple un PDA ou encore un réfrigérateur intelligent. Nous appellerons ces derniers des dispositifs. Ils sont fournis par des constructeurs et, pour la plus grande majorité d'entre eux, ne sont **pas prévus pour être modifiables** ni physiquement, ni logiciellement. A partir de l'ensemble des dispositifs accessibles il est possible pour une entité de créer de nouvelles applications en faisant interagir les dispositifs les uns avec les autres [48]. Nous appellerons cet ensemble des dispositifs et entités logicielles auxquels un système peut accéder son **infrastructure**. Ces dispositifs peuvent être **mobiles** [51, 46], ils peuvent entrer ou quitter à tout instant cette infrastructure. Lorsque c'est le cas, les applications doivent alors s'adapter à ces variations [48]. Cette infrastructure impose alors les contraintes suivantes aux systèmes ambiants :

Hétérogénéité : l'ensemble des dispositifs qui compose l'infrastructure d'un système ambiant peut être hétérogène aussi bien en terme de matériel que de communications. Il faut être capable de les faire interagir les uns avec les autres.

Multiplicité des entités se trouvant dans l'environnement : Un grand nombre et de nombreuses variantes d'entités et de phénomènes peuvent se trouver dans l'environnement d'un système ambiant et doivent être considérés. Le système ambiant peut alors lui-même prendre de nombreuses formes.

Briques logicielles de base non-éditables : les dispositifs fabriqués et commercialisés par des constructeurs ne sont généralement pas prévus pour être modifiés physiquement ni logiciellement. La création de nouvelles fonctionnalités dans un système ambiant ne peut donc pas passer par la modification de ces dispositifs.

Mobilité : avec l'apparition des dispositifs et logiciels mobiles, les déconnexions ne sont plus considérées comme des pannes mais comme un mode de fonctionnement normal [94]. En raison de cette mobilité des dispositifs qui composent l'infrastructure d'un système, la topologie de cette dernière est fortement variable.

Ajouté au besoin de s'adapter aux variations de leur infrastructure, les systèmes ambiants, dans l'optique de s'insérer de manière transparente dans leur environnement, doivent également prendre en compte ses évolutions et s'y adapter [51]. L'adaptation se définit dans le dictionnaire comme le fait d'« *ajuster une chose à une autre* » [1]. Dans le domaine du logiciel, on la définit comme : « *le processus de modification du système, nécessaire pour permettre un fonctionnement adéquat dans un contexte donné* » [145]. On considère qu'une adaptation est nécessaire quand il n'y a plus de correspondance entre l'offre et la demande à une ressource [51]. Dans le cadre de l'informatique ambiante, comme nous l'avons vu précédemment, la forte variabilité de l'environnement et de l'infrastructure d'un système ambiant ont pour conséquence le besoin d'adaptation. Cela peut être pour garantir une bonne continuité de service, ou encore pour faire correspondre au mieux le comportement du système à son environnement.

Pour cela, le système doit être sensible aux variations de son environnement, aux variations de son contexte, afin de modifier la façon dont il s'exécute. Il s'agit du moyen pour le système de s'adapter à une situation en restant le moins intrusif possible pour l'utilisateur. La notion de contexte fut utilisée

1. Dictionnaire Littré

en informatique, en premier lieu, dans des domaines comme l'intelligence artificielle ou encore la théorie des langages avant de connaître un regain d'intérêt avec l'essor de l'informatique ambiante. Le contexte devient alors une notion centrale pour la plupart des travaux d'informatique ambiante et de nombreuses définitions émergent sans que cela n'aboutisse réellement à un consensus. En 2001, Dey pose la définition du contexte la plus couramment utilisée : « *Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves.* » [37]. S'adapter au contexte nécessite alors la mise en place d'un cycle de collecte puis d'évaluation de ces informations contextuelles pour la mise en place de réponse à ces changements de contexte. Une autre caractéristique des systèmes ambiants vient alors s'ajouter aux précédentes :

Sensibilité au contexte : Un système ambiant se trouve dans un environnement qui est en perpétuelle évolution. Les ressources disponibles, qui peuvent être mobiles, ou encore les phénomènes physiques observés, changent sans cesse et, chacun avec sa propre dynamique.

Ces caractéristiques, lorsqu'elles sont combinées les unes avec les autres, sont à l'origine d'un ensemble de challenges et soulèvent les questions que nous aborderons dans cette thèse. Le premier challenge est relatif à la variabilité et l'imprévisibilité du contexte d'un système ambiant. Les multiples dispositifs potentiellement hétérogènes et mobiles qui composent l'infrastructure d'un système ambiant, ainsi que les divers phénomènes perpétuellement changeant pris en compte dans l'environnement, mènent à la conception de systèmes qui doivent être capables de gérer trois axes de variabilité.

1.2.1 Trois grands axes de variabilité

L'environnement d'un espace ambiant est caractérisé par une grande variabilité [100]. Ces variations s'expriment en particulier selon trois axes.

1. *Le premier axe de variabilité concerne les variations de l'ensemble des dispositifs disponibles dans l'infrastructure du système*. Nous avons vu, en effet, que les applications d'informatique ambiante reposent sur une infrastructure susceptible d'englober une multitude de dispositifs. Ces derniers peuvent subir des pannes, ne plus être accessibles, ou encore être mobiles. Ils sont alors en mesure d'apparaître ou de disparaître dans cette infrastructure. Il ne s'agit plus d'événements exceptionnels mais normaux dans la vie d'un système ambiant [48]. Par conséquent, l'application qui sera construite au dessus de celle-ci devra varier en fonction des opportunités offertes.
2. *Le second axe de variabilité qu'il est nécessaire de prendre en compte pour un système ambiant concerne la haute variabilité de l'environnement d'une application.* Par nature, l'environnement est en perpétuel changement [45] ; il est un espace continu dans lequel des phénomènes physiques interviennent et évoluent continuellement. La prise en compte de ces changements se fera via son observation. Les variations s'expriment alors sous la forme de changements de valeurs ou d'états, associées à des modèles physiques ou encore informatiques (par exemple modèle et seuils pour des températures). Afin de mesurer et de raisonner sur ces phénomènes, le système doit posséder ces modèles [38].
3. *Le troisième axe de variabilité porte sur l'ensemble des adaptations, pour des préoccupations diverses, qui doivent être mises en œuvre*. En fonction des deux premiers axes de variabilité, c'est-à-dire en fonction de son environnement et de son infrastructure, une application pourra prendre plusieurs configurations. Le mécanisme d'adaptation est l'outil permettant de passer

d'une configuration à une autre. Différentes adaptations peuvent être décrites pour diverses préoccupations et peuvent être réalisées ensembles lorsqu'une nouvelle situation l'exige. Ce dernier axe est fortement dépendant des deux autres, puisqu'il consiste en la réaction à leurs variations. En effet, l'ensemble des configurations atteignables par une application dépendra en parti de l'étendue de la gestion des variations de l'environnement et de l'infrastructure. Plus le système peut gérer de variantes, plus il est susceptible d'avoir à gérer de nombreuses configurations de l'application. L'ensemble des adaptations qui peuvent être mises en œuvre pour passer d'une configuration à une autre peut alors rapidement devenir conséquent et ainsi faire exploser le nombre de combinaisons d'adaptations [49].

Ces trois axes de variabilité soulève une question que nous nous poserons dans ce manuscrit : **Comment définir un mécanisme d'adaptation permettant d'exprimer la grande variabilité du système, de son infrastructure, de son environnement, tout en minimisant le nombre d'entités d'adaptation à écrire ?**

Dans le cycle de sensibilité au contexte, la prise en compte de ces variabilités se fait lors de l'évaluation des informations contextuelles par un mécanisme de décision en vue de mettre en œuvre les réponses adaptées. Ce mécanisme de décision peut alors spécifier, pour l'ensemble de variantes des deux premiers axes, l'ensemble des adaptations qui doivent être réalisées et de quelle manière. Lorsque l'on se trouve dans un environnement dit borné [45], c'est-à-dire impliquant un ensemble limité, parfois même fixe, de dispositifs, et par conséquent permettant une perception limitée du contexte, il peut être envisagé de concevoir un système ambiant en spécifiant toutes les variantes des trois axes précédemment évoqués. L'environnement peut être considéré comme connu a priori. Spécifier toutes ces variantes permet de concevoir des systèmes avec une fiabilité maximale. Malheureusement dans des environnements non bornés [49], qui sont a priori non connus, cela n'est plus possible. En effet, l'ensemble des dispositifs gravitant autour d'un système n'étant alors ni fixe, ni limité, la combinatoire des trois axes de variabilité est susceptible de devenir trop grande pour être spécifiée [49] ou ne peut tout simplement pas l'être. Il est impossible d'en prédire l'ensemble des variations, et, parfois, des variantes peuvent ne pas avoir été anticipées. De la même manière, plusieurs acteurs peuvent déployer de nouvelles adaptations sans se connaître.

1.2.2 Imprévisibilité et combinatoire dans des environnements non bornés

A la notion de variabilité vient donc s'ajouter celle d'imprévisibilité. Pour chacun de ces éléments variables, de nombreuses informations peuvent ne pas être prévues à la conception et doivent être gérées de manière non-anticipée. La notion d'imprévisibilité vient donc étendre celle de variabilité sur ces trois axes précédemment définis.

1. *En raison de la mobilité ou de pannes, il n'est pas possible de connaître, parmi les dispositifs que l'on sait utiliser, lesquels sont disponibles à un instant donné*. Si nous souhaitons utiliser des dispositifs dans un but particulier il faut connaître les capacités qu'ils offrent ; ce point ne peut pas être sujet à imprévisibilité. Par contre, on ne peut pas formuler l'hypothèse de leur disponibilité.
2. *On doit connaître les phénomènes que l'on mesure dans l'environnement et comment les analyser, mais on ne sait pas a priori comment ces derniers vont évoluer*. Par exemple, il est possible d'observer le comportement humain mais il est impossible de le prédire avec certitude. D'autre part, le système peut accéder aux données sur son environnement grâce à son infrastructure, sa perception en est donc limitée aux capacités offertes par cette infrastructure, ces données ne

sont alors que partiellement accessible [45]. Cette perception évolue en fonction de la disponibilité des dispositifs et donc de manière imprévisible. Enfin, l'imprévisibilité de l'environnement peut parfois mener à des situations dans lesquelles, le résultat de l'interprétation d'informations contextuelles est incertain.

3. *Le système de prise en compte du contexte connaît donc des reconfigurations pour un but mais ne sait pas, quand et avec qui elles doivent être composées*. Dans les deux cas précédents, l'imprévisibilité vient pour partie du fait que le système n'a pas le contrôle sur les variations ; elles lui sont imposées. Ce n'est pas le cas de l'imprévisibilité dans le cadre du troisième axe de variabilité, c'est-à-dire par rapport à l'ensemble des adaptations à mettre en œuvre à un instant donné. L'imprévisibilité, ici, est héritée de celle des deux autres axes. Le passage d'une configuration du système à une autre est réalisé par des adaptations. Une adaptation étant réalisée pour un but particulier, plusieurs adaptations peuvent potentiellement intervenir lors d'une même reconfiguration. Ainsi, toutes les configurations ne peuvent pas être prévues *a priori* puisque reposant sur des niveaux imprévisibles, la réaction à des événements imprévus est elle-même par définition imprévue.

Nous avons vu que pour chacun de ces axes, différentes informations doivent être connues à l'avance. Cependant l'ensemble de ces informations ne peut être exhaustivement anticipé à la conception. Lorsque l'on souhaite modifier cet ensemble d'informations (ajouter des modèles de phénomènes physiques, des descriptions de dispositifs, des adaptations), cela doit pouvoir être réalisé à l'exécution sans interrompre l'application. On parle d'**extensibilité**. Mais en raison de la combinatoire ainsi que de la variabilité de ces éléments connus, il faut être capable de rajouter ou de supprimer ces informations sans tenir compte de celles existantes. Des mécanismes de décision plus ou moins aptes à prendre en compte ces trois axes d'imprévisibilité existent et sont adaptés aux différents types de systèmes ambiants et d'environnements. Les environnements les plus bornés ne nécessitent pas de prendre en compte l'imprévisibilité contrairement aux environnements ouverts. Les questions que nous nous posons sont alors : **Comment associer aux différents mécanismes de décision un mécanisme d'adaptation que tous peuvent utiliser, c'est-à-dire permettant de déployer des adaptations sans se préoccuper de celles existantes ou de les combiner explicitement ? Comment permettre le déclenchement des adaptations indépendamment les unes des autres ?**

Nous avons vu que l'environnement et l'infrastructure d'un système ambiant varient fortement et d'une manière qui ne peut pas être anticipée à la conception. Ces variations évoluent en respectant des dynamiques qui leurs sont propres. Les mécanismes de sensibilité au contexte, qui prennent en considération ces variations, doivent être capables de respecter au mieux les dynamiques d'évolution des phénomènes variables auxquels ils s'intéressent.

1.2.3 Dynamicité et multi-dynamicité

Les mécanismes de sensibilité au contexte jouent un rôle de médiateur entre les applications et leur environnement [100]. Par conséquent, lorsqu'ils mettent en œuvre une adaptation, ils doivent respecter deux grandes dynamiques d'évolution : celle de l'application à adapter et celle de leur environnement. En effet, l'environnement évolue perpétuellement et le système doit répondre à ces sollicitations mais aussi à celles de l'application (et bien entendu de l'utilisateur). Si l'environnement peut être vu comme une unique entité évoluant suivant une dynamique, il apparaît en réalité que celui-ci peut varier selon divers axes, qu'il s'agisse de phénomènes physiques ou de l'infrastructure sur laquelle repose l'application. Ainsi des phénomènes physiques comme la luminosité d'une

pièce ou encore le nombre d'utilisateurs qui se trouvent à l'intérieur, évoluent selon leurs propres dynamiques. Un système ambiant doit donc être capable de respecter du mieux qu'il peut plusieurs dynamiques d'évolution.

Ainsi, en raison du besoin de sensibilité au contexte et de la mobilité des dispositifs composant l'infrastructure d'une application, le mécanisme de prise en compte de l'environnement, externe à l'application, doit respecter les dynamiques présentées précédemment. Plus particulièrement, il doit, a minima, respecter une dynamique d'évolution : celle de l'infrastructure de l'application. C'est seulement grâce aux dispositifs se trouvant dans cette infrastructure que l'application peut être construite et intégrer de nouvelles fonctionnalités. Il faudra alors s'adapter à toutes ces différentes variations du contexte et de l'infrastructure. Pour ce faire, plusieurs mécanismes de sensibilité au contexte et par conséquent de mécanismes de décision peuvent être utilisés. Les questions que nous nous posons sont alors : **Comment organiser ces mécanismes de décision ? Comment prendre en compte les multiples dynamiques d'évolution de l'environnement ? Comment prendre en compte ces multiples évolutions en interrompant le moins possible le système ambiant ?**

Afin de pouvoir respecter ces dynamiques, la fréquence des adaptations autorisées par le système doit être adaptée. L'application doit fournir des fonctionnalités qui sont cohérentes avec son environnement et ne pas se trouver dans un état instable ou stoppé en permanence à cause de réadaptations continues.

1.2.4 Des temps de réponse adaptés et maîtrisés

Dans ce cadre, la fréquence à laquelle un mécanisme d'adaptation peut enchaîner les adaptations est une préoccupation majeure en informatique ambiante. En effet, les contraintes précédemment évoquées mènent à la mise en place de systèmes complexes. Nous allons voir que **la pertinence de l'adaptation ne doit pas seulement être logique mais aussi temporelle.** Considérons que les applications adaptables sont toujours dans un des trois états présentés en FIGURE 2.12.

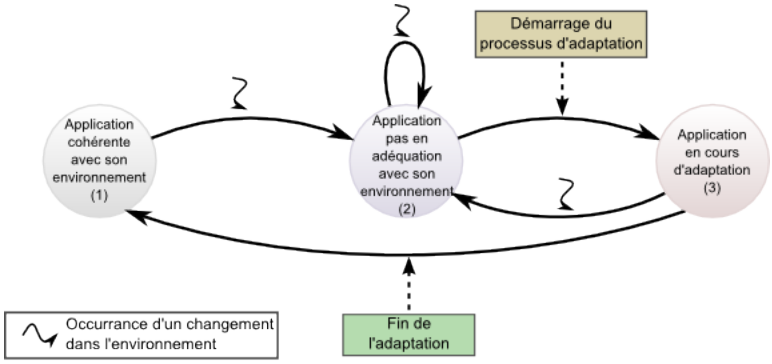


FIGURE 1.1 – Les trois états d'une application adaptable.

L'état (1) est l'état d'exécution normale de l'application pour lequel l'application est censée être consistante avec son environnement. Cela signifie que le comportement de l'application repose sur ce qui est pertinent dans son environnement et qu'il s'agit bien du comportement souhaité pour cette situation. Lorsqu'un changement intervient dans le contexte de l'application et avant d'être adaptée, celle-ci se trouve alors dans un état (2) dans lequel elle n'est plus consistante avec l'environnement.

Une fois l'adaptation démarrée, elle se trouve dans un état instable (3) et peut être tout ou en partie indisponible. Elle n'est également plus consistante avec son environnement. Pendant que l'application est dans l'état (2) plusieurs nouvelles modifications de l'environnement peuvent avoir lieu. Si un de ces changements intervient pendant que l'application est dans l'état (3), alors celle-ci retombe dans l'état (2). Le temps passé dans les états (2) et (3) doit être géré de manière à correspondre au mieux aux deux types de dynamiques : (a) celle de l'environnement et (b) celle de l'application. Dans le premier cas, la dynamique de l'adaptation doit être compatible avec celle des changements intervenants dans l'environnement (a). Pour cela, il est essentiel de considérer les cas suivants :

- Il ne faut pas laisser l'application trop longtemps indisponible (3) ou dans l'état qui ne convient plus (2). Par exemple, si des dispositifs deviennent indisponibles, il ne faut pas que l'application continue à utiliser les fonctionnalités qu'ils proposent ou encore que l'utilisateur pense encore pouvoir les utiliser. La latence entre deux adaptations doit être suffisamment faible pour permettre d'adapter l'application avec une fréquence au plus proche de celle de l'environnement.
- D'autre part, une trop forte latence entraînerait le risque de sombrer dans un système instable qui consisterait à ne jamais atteindre l'état suivant (1) avant une nouvelle évolution de l'environnement. L'application sombrerait alors dans une forme d'indéterminisme dont elle ne saurait sortir, c'est-à-dire ne plus connaître l'état de l'application et comment elle va être adaptée, ceci est mis en avant par le cycle entre les états (2) et (3).

Dans le second cas (b), la dynamique de l'adaptation doit être compatible avec les opérations de l'application et/ou de l'utilisateur. Pour cela, il est essentiel de prendre en compte les points suivants :

- Le système ne doit pas passer de l'état (1) à (3) trop fréquemment et produire une application instable. L'utilisateur serait alors dans l'incapacité d'utiliser l'application, et celle-ci pourrait alors passer plus de temps dans l'état (3) que dans des états stables.
- La latence du système ne doit pas être trop grande au risque de détourner l'utilisateur, ou qu'il n'utilise pas le système de façon prévue [146].

Le déclenchement de l'adaptation dépend du mécanisme d'observation du contexte, il est donc également nécessaire que ce dernier offre une faible latence. Pour ce faire, ces mécanismes peuvent s'exprimer en parallèle et, pour que l'environnement impose son rythme, ils peuvent reposer sur une approche de type push, c'est-à-dire ne pas requérir des informations sur l'environnement mais recevoir les informations de l'environnement. La problématique des **temps de réponse**, qui doivent être maitrisés et adaptés, sera présente tout au long de ce manuscrit. A partir de l'ensemble des caractéristiques, défis et questions que venons de soulever et qui seront synthétisés dans la section suivante, nous définirons les objectifs de cette thèse.

1.3 Synthèse et objectif

La FIGURE 1.2 présente une synthèse des caractéristiques de l'informatique ambiante et des challenges auxquels nous nous intéresserons dans ce manuscrit.

Dans cette thèse, nous chercherons tout d'abord à identifier dans la littérature les approches, sur lesquelles peuvent reposer les mécanismes d'adaptation, qui nous permettront d'apporter des solutions aux challenges que nous avons présentés précédemment. En particulier, nous nous intéresserons aux techniques permettant la gestion de la variabilité du système tout en garantissant une faible interruption de son exécution. Ces adaptations sont ensuite utilisées par des mécanismes, appelés de décision, qui déterminent lesquelles d'entre elles doivent être mise en œuvre et de quelle manière. Ils déterminent en partie les capacités d'un système à gérer la variabilité et l'imprévisibilité. Nous

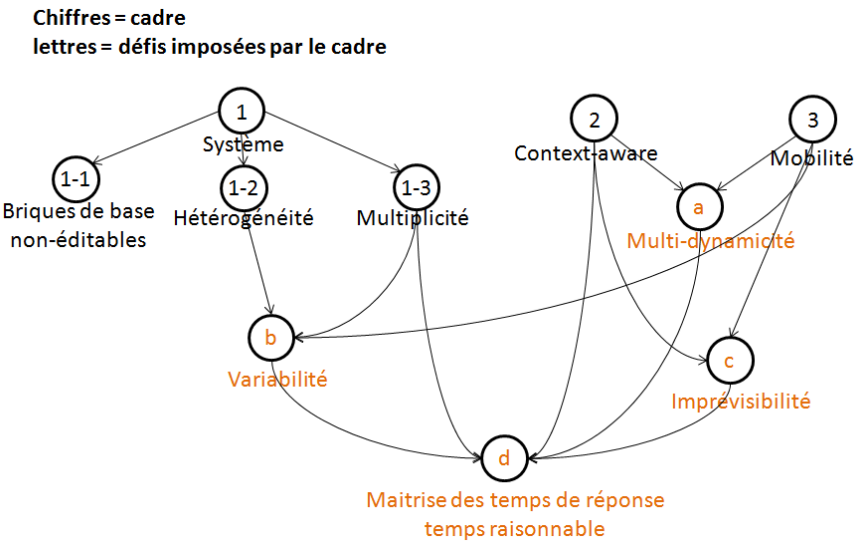


FIGURE 1.2 – Caractéristiques de l'informatique ambiante.

observerons alors que de nombreuses approches existent pour définir de tels mécanismes. Nous identifierons ainsi un continuum entre les approches les plus aptes à prendre en compte l'imprévisibilité de l'environnement et les approches qui ne traitent pas cette problématique. Nous verrons que selon les objectifs du système et l'environnement dans lequel il se trouve, chaque approche peut avoir du sens. Par exemple, un système limitant fortement sa prise en compte de phénomènes qui n'avaient pas pu être anticipés à la conception, doit permettre la mise en place d'un système plus fiable. Le chapitre 2 détaillera cet état de l'art.

A partir de ces constats, nous proposerons dans le chapitre 3 une architecture permettant de respecter les multiples dynamiques de l'environnement y compris celles de l'application, de l'infrastructure et de l'utilisateur. Cette architecture permettra également de minimiser les temps de blocage de l'application. De plus il autorisera l'utilisation de différents types de mécanismes de sensibilité au contexte du plus rigide au plus flexible. Il proposera également une organisation de ces mécanismes. Afin qu'il soit possible de déployer dans cette architecture différents types de mécanismes de décision, il faut lui associer un mécanisme d'adaptation que tous puisse utiliser.

Le chapitre 4 propose un tel mécanisme. Nous identifierons pour ce mécanisme un ensemble de propriétés logiques et temporelles qui nous permettra de vérifier les caractéristiques que nous avons identifiées dans l'état de l'art et qui pourra être utilisé par les différents types de mécanismes de décision. Nous proposerons alors une mise en œuvre de ce mécanisme sur lequel, afin de vérifier les propriétés temporelles, nous réaliserons des évaluations de performances sur les temps de réponse qu'il propose.

Nous proposons ensuite, dans le chapitre 5, une mise en œuvre de notre modèle architectural ainsi que de notre mécanisme d'adaptation. Elle sera appliquée à un scénario se déroulant dans le cadre d'un bâtiment intelligent. Nous présenterons alors une implémentation de chacun des niveaux de notre architecture.

Enfin, dans la dernière partie du manuscrit, un bilan des travaux sera dressé, et les perspectives pour les travaux futurs seront décrites.

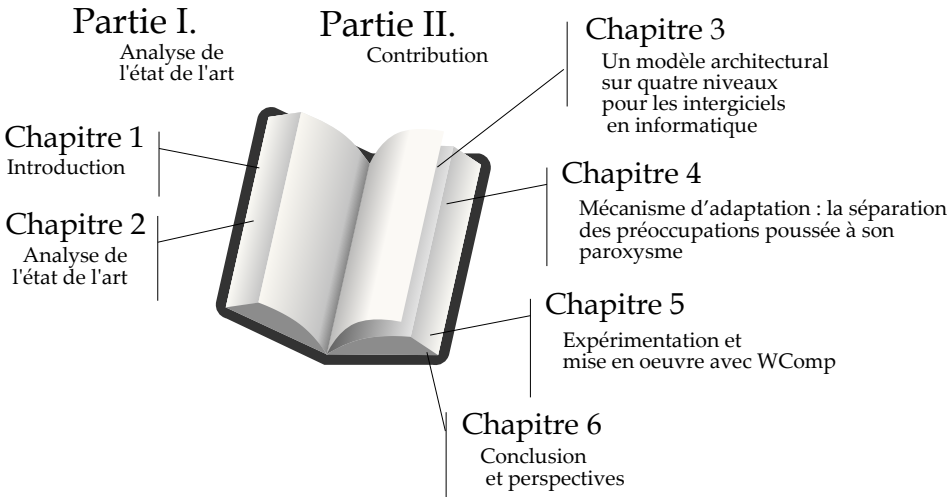


FIGURE 1.3 – Plan du manuscrit.

CHAPITRE 2

Analyse de l'état de l'art sur les mécanismes d'adaptation au contexte en informatique ambiante

« Le désordre est bien puissant quand il s'organise. »

André Suarès.

Sommaire

CE CHAPITRE présente différentes approches sur lesquelles peuvent reposer les mécanismes d'adaptation ainsi qu'un continuum d'approches, plus ou moins flexibles, pour la sélection d'adaptations en fonction du contexte.

Dans un premier temps, nous étudierons les approches relatives aux mécanismes d'adaptation. En fonction des contraintes imposées par notre cadre de travail, l'informatique ambiante, nous déterminerons les caractéristiques qu'un mécanisme d'adaptation doit respecter ainsi que les approches qu'il

doit utiliser. Dans cet objectif, nous identifierons tout d'abord, le besoin de disposer d'un mécanisme permettant de réaliser des **adaptations dynamiques**. En effet, lorsque l'on souhaite être sensible au contexte qui, par nature, est fortement variable, il n'est pas envisageable de stopper puis redéployer l'application à chaque adaptation. Nous présenterons par la suite les deux grands types d'adaptations dynamiques. Nous constaterons alors la nécessité de réaliser des **adaptations compositionnelles**, c'est-à-dire permettant d'intégrer dynamiquement de nouveaux algorithmes dans une application. En particulier, ce type d'adaptation offre au système la capacité d'exploiter dynamiquement de nouvelles fonctionnalités logicielles offertes par des dispositifs qui viendraient d'apparaître.

De nombreuses modifications peuvent être portées dans l'application lors d'adaptations, perturbant ainsi son fonctionnement, et ce, potentiellement avec des effets néfastes. Nous verrons alors qu'il est possible de réaliser ces adaptations *a priori* sur une **représentation abstraite** de l'application. Il est alors possible d'évaluer l'impact des adaptations avant même de les reporter dans l'application. Enfin, nous observerons que l'adaptation est une préoccupation transverse qui, dans les approches de programmation classiques, qu'elles soient orientées objets, composants ou encore services, peut difficilement être encapsulée dans une entité réutilisable. Nous étudierons alors quelques approches permettant de leur appliquer le principe de la **séparation des préoccupations**, dissociant ainsi clairement les préoccupations d'adaptations de la logique applicative.

Lorsque des adaptations séparées dans des modules sont appliquées, elles doivent être composées. Puisque nous souhaitons permettre à des systèmes de s'adapter à leur contexte, l'objectif des mécanismes d'évaluation du contexte est de sélectionner ou déclencher des adaptations et ainsi de définir lesquelles doivent être **composées**. Parfois, ces mécanismes spécifient, dans une approche explicite et rigide, de quelle manière composer toutes les adaptations les unes par rapport aux autres. Dans un second temps, ce chapitre présentera donc différentes approches d'évaluation du contexte. Ces mécanismes, en fonction de la fléxibilité avec laquelle ils définissent « quand » et « comment » doivent être appliquées les adaptations, sont plus ou moins aptent à gérer l'imprévisibilité de l'environnement du système. Nous identifierons, pour chacun d'entre eux, les caractéristiques que doit respecter le mécanisme d'adaptation qui lui est associé.

2.1 De l'adaptation statique à l'adaptation dynamique

Une application peut être adaptée lors de diverses phases de son cycle de vie. Cependant, en raison de la forte variabilité du contexte et afin de conserver la pertinence temporelle des adaptations, il est important de réaliser ces adaptations de manière à interrompre le moins possible l'exécution de l'application.

Une adaptation est qualifiée de *statique* lorsqu'elle intervient avant l'exécution [135], c'est-à-dire avant que l'application ne soit déployée [77]. Ce type d'adaptation nécessite, par conséquent, l'arrêt de l'application et son redéploiement (FIGURE 2.1). Si l'adaptation statique ne nécessite pas de s'appliquer sur une application qui a été prévue pour [25], elle requiert que les contraintes nécessitant l'adaptation varient peu, de telle manière que lorsque l'adaptation est déployée, elle peut rester stable suffisamment longtemps [135]. Des mécanismes génériques pour réaliser de telles adaptations existent, comme par exemple, AspectJ [14]. A l'origine, avec AspectJ, l'adaptation intervient lors de la phase de compilation. Le code décrit dans des entités abstraites est injecté dans le code de l'application pour générer un nouveau fichier source. Nous avons vu dans l'introduction

que les systèmes ambiants doivent respecter plusieurs dynamiques d'évolution. La fréquence de ces évolutions pouvant être élevée, il n'est pas envisageable d'arrêter, puis d'adapter pour enfin redéployer l'application à chaque fois que la situation exige une adaptation [54]. Nous souhaitons, au contraire, garder l'application utilisable le plus longtemps possible.

L'adaptation dite *dynamique* permet de modifier, totalement ou partiellement, le comportement d'un logiciel (OS, intergiciel, application ...) tandis que celui-ci continue à s'exécuter [144]. Contrairement à l'adaptation statique, il faut alors que l'application soit prévue pour cela, c'est-à-dire à minima construite sur une plateforme permettant son adaptation. Une solution pour répondre à cette problématique est de combiner approches statiques et dynamiques. Par exemple, Yang *et al* [25] proposent de permettre à des applications qui n'ont pas été conçues pour cela de s'adapter dynamiquement. Un processus en deux phases est alors réalisé. Cette approche consiste, dans un premier temps, à adapter statiquement l'application pour y ajouter des mécanismes qui permettront lors d'une seconde phase son adaptation dynamique par insertion ou suppression de code. Ce type d'approche est difficilement envisageable dans le cadre de l'informatique ambiante. En effet, en raison de l'imprévisibilité des évolutions du contexte d'une application ubiquitaire, il est impossible de prédire, à la conception, l'ensemble des adaptations qui pourront être réalisées sur une application. Dans les approches couplant adaptation statique et dynamique, cela aurait pour effet d'entraîner régulièrement la mise en œuvre de l'adaptation statique pour mettre à jour les capacités d'adaptation dynamique.

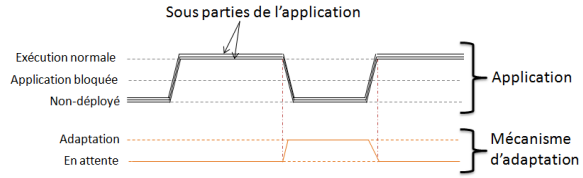


FIGURE 2.1 – Adaptation statique

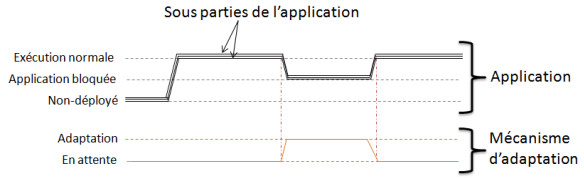


FIGURE 2.2 – Adaptation dynamique bloquante

On observe plutôt dans la littérature un continuum d'approches dynamiques pour lesquelles l'application est plus ou moins bloquée durant leur mise en œuvre. Dans le cas le plus contraignant, l'adaptation se fait à l'exécution mais a pour conséquence de rendre l'application entièrement indisponible pendant la durée de l'adaptation (FIGURE 2.2). Il n'y a pas de phase de redéploiement mais elle est bloquée dans un état dit « quiescent » [144], c'est-à-dire un état pour lequel l'adaptation est sûre [82]. Par exemple, cela peut se faire en permettant aux nouveaux nœuds de l'application d'être initialisés de manière à être cohérents avec le reste de l'application. Il s'agit de l'approche proposée dans le projet Mobile Users in Ubiquitous Computing Environments (MUSIC) [100]. Dans ce dernier, une application prend la forme d'un assemblage de composants. Un moteur de décision a pour but de sélectionner la configuration d'une application adaptée à son contexte. Avant de déployer la nouvelle application, l'entité appelée le *component configurator* place l'application dans un état quiescent en suspendant l'exécution de ces composants. S'il garantit le bon déroulement de l'adaptation, ce type d'approche nuit à la continuité de service du système. Pendant l'adaptation, l'application toute entière n'est plus utilisable alors que des parties de celle-ci ne sont pas affectées par les modifications. Dans le continuum des approches que l'on retrouve dans la littérature, d'autres approches sont moins bloquantes et proposent de circonscrire ce phénomène à la sous-partie de l'application impactée par l'adaptation.

Comme cela est noté dans [144], l'application n'a pas forcément besoin d'être entièrement bloquée pendant son adaptation : « *It should not be necessary to stop the whole of a running application system to modify part of it. The management system should, from the change specification, be able to determine a minimal set of nodes which are affected by the change. The rest of the system should be able to continue its execution normally* ». Il peut être possible de stopper uniquement la sous-partie de l'application ciblée par l'adaptation (FIGURE 2.3), voir a minima de bloquer celle-ci seulement pendant l'instruction atomique de modification (Figure 2.4). Il s'agit là de la plus petite unité de blocage, l'adaptation totalement dynamique, c'est-à-dire ne nécessitant aucun arrêt de l'application, n'étant pas envisageable. En effet, il y a toujours un temps minimal pendant lequel tout ou partie de l'application est partiellement stoppée.

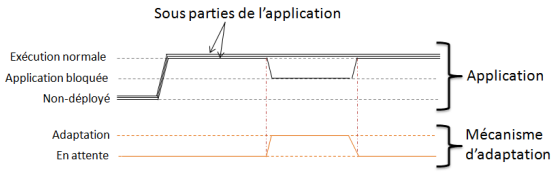


FIGURE 2.3 – Adaptation dynamique partiellement bloquante

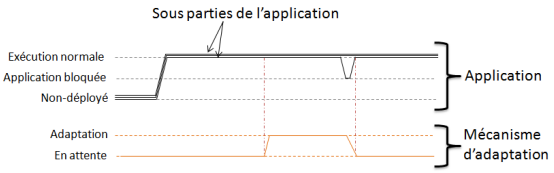


FIGURE 2.4 – Adaptation dynamique la moins bloquante possible

Nous nous orientons vers l'adaptation la plus dynamique possible car il n'est pas envisageable de redéployer ou de bloquer entièrement une application à chaque apparition ou disparition de dispositifs comme à chaque changement de contexte. Au contraire, nous souhaitons garder l'application la plus utilisable possible pour une meilleure continuité de service et minimiser les temps d'adaptation de l'application.

Maintenant que nous avons identifié, parmi les différentes phases pendant lesquelles il est possible d'adapter une application, celle qui nous intéresse, nous allons identifier sur quelles entités et de quelle manière ces adaptations doivent être réalisées. Pour cela, deux grandes approches d'adaptation dynamique émergent dans la littérature.

2.2 Adaptations compositionnelles et paramétrées : des approches qui peuvent être combinées

Deux approches principales permettent de réaliser des adaptations dynamiques [95] : (1) l'adaptation paramétrée et (2) l'adaptation compositionnelle. La première modifie le comportement de l'application en faisant varier des paramètres existants, tandis que la seconde permet de reconfigurer une application en ajoutant, retirant, échangeant les modules qui la composent. Ces approches, loin d'être incompatibles, offrent des capacités qu'il peut être intéressant de combiner. Nous allons maintenant étudier ces deux approches plus en détail et plus particulièrement en terme de temps de réponse et de capacité à étendre les fonctionnalités du système.

2.2.1 L'adaptation paramétrée

L'adaptation paramétrée a pour objectif de modifier des variables afin de changer le « comportement » d'une application. Les paramètres peuvent être modifiés à l'exécution et entraîner la mise en

place d'une nouvelle stratégie, qui avait été prédéfinie, dans l'application. Cette approche est simple d'utilisation et offre des temps de réponse faibles. Cependant, l'ensemble des adaptations réalisables est borné [58] et doit être défini à la conception. Les systèmes réalisant de telles adaptations sont parfois appelés « *closed-adaptive* » [152], puisqu'ils choisissent les adaptations parmi celles disponibles sans qu'il soit possible d'en ajouter à l'exécution. Cette approche est donc bien adaptée aux applications se trouvant dans des environnements bornés, pour des buts bien précis et maîtrisés, pour lesquels un concepteur peut anticiper et spécifier l'ensemble des adaptations.

La Context Toolkit [73] utilise une telle approche. Il s'agit d'une infrastructure de référence dans le domaine de la capture du contexte. Elle reprend la métaphore de la boîte à outil graphique et propose des *context widgets* qui font la liaison entre une application et son environnement. Une des évolutions de cette boite à outils est l'apparition d'un composant appelé *enactor* qui permet de décrire une logique applicative. Un *enactor* est en capacité d'exposer un certain nombre de paramètres accessibles selon différents modes (écriture, lecture ...). Ces paramètres définissent l'interface publique de l'*enactor* et sont analogues aux propriétés et paramètres des composants classiques. Chacun d'entre eux possède une description informant de son type et de son influence sur le bloc applicatif [151]. Ces paramètres peuvent être modifiés dynamiquement pour changer le comportement du bloc applicatif associé ou se trouvant dans l'*enactor*. Une fois le comportement modifié, une notification décrivant le changement est envoyée aux abonnés.

Ce type d'adaptation peut être utilisé dans des cas bien précis qui nécessitent des temps de réponses minimes. Cependant, parce que l'on ne peut imaginer qu'il soit possible d'anticiper tous les cas d'adaptation, aussi bien dans la partie logicielle modifiable du système, que sur les dispositifs eux-mêmes, l'utilisation unique de l'adaptation paramétrée dans le cadre de l'informatique ambiante est difficile à concevoir. En effet, comme nous l'avons vu, l'adaptation paramétrée requiert la spécification à la conception de tous les paramètres et de tous les algorithmes définissant le comportement d'une application.

2.2.2 L'adaptation compositionnelle

L'adaptation compositionnelle [95] vise à échanger des algorithmes ou stratégies d'une application par d'autres qui correspondent mieux aux besoins du système. Une adaptation peut être conçue et intégrée plus tard, pendant l'exécution, par *late-binding*. Comme nous pouvons le voir dans la littérature, l'adaptation compositionnelle est donc particulièrement adaptée au cadre de l'IAm et à la gestion des variations de l'infrastructure logicielle d'une application [114, 38, 53]. L'adaptation compositionnelle offre ainsi la possibilité d'intégrer dans l'application de nouveaux algorithmes qui n'avaient pas été prévus à la conception [58]. Les systèmes réalisant de telles adaptations sont parfois appelés « *open-adaptive* » [152]. En Informatique ambiante, puisque nous possédons un ensemble de briques logicielles fournies par des dispositifs qu'il n'est pas nécessairement possible de modifier, nous souhaiterons les faire interagir entre elles dans l'optique de créer de nouvelles fonctionnalités [90, 60]. Pour ce faire, l'adaptation doit être capable de prendre en compte les apparitions ou disparitions de ces dispositifs. Alors, comme cela est écrit dans [60] : « *The ability to seamless compose services from various devices in a more or less ad-hoc manner is a frequently emphasized feature of ubiquitous computing* » . Dans ce cadre, l'adaptation compositionnelle permet, par exemple, de prendre en compte dynamiquement les parties logicielles exposées par des dispositifs qui viennent d'apparaître et inversement lors de leur disparition. Par contre, les temps de réponses engendrés sont plus importants que ceux de l'adaptation paramétrée, les mécanismes mis en œuvre

étant plus complexes. Quoiqu'il en soit, ces deux types d'adaptations ne sont pas incompatibles. Dans les projets MADAM [76] et MUSIC [100], ces deux types d'adaptations sont mis en œuvre avec un focus plus particulier sur l'adaptation compositionnelle. Elle est utilisée afin de modifier des configurations d'assemblages de composants. Dans la littérature, de nombreux intergiciels [54, 57, 21, 68, 98, 107, 67] proposent d'appliquer l'adaptation compositionnelle au-dessus de plateformes d'exécution à composants.

Par nature, pour pouvoir réaliser des adaptations compositionnelles d'un logiciel, il faut que celui-ci repose sur une architecture modulaire et qu'il y ait un couplage faible entre les entités logicielles que l'on adapte [95]. Si les approches orientées objets offrent une modularité, elles n'ont pas réussi à apporter une solution bien établie sur la manière d'assembler ultérieurement les entités logicielles [64]. Les approches à services ou composants offre cette possibilité.

2.2.2.1 Architectures orientées services

Les services sont des entités logicielles orientées *business* et non pas technique comme peuvent l'être les composants. Par contre, à l'instar des composants, les services sont vus par leurs consommateurs comme des boites noires. Un même service peut être découvert et utilisé par plusieurs consommateurs. Pour ce faire, un service expose une interface publique parfois appelée contrat. Afin de conserver leur propriété de boite noire, cette interface ne contient aucune information relative à leur implémentation. Un service doit se conformer à son contrat puisqu'il s'agit là du seul moyen pour les autres entités d'obtenir des informations sur ses fonctionnalités. Les services, qui sont des entités faiblement couplées, peuvent être coordonnés pour s'assembler et créer de nouvelles applications ou des services composites. On parle alors d'architectures orientées service (SOA). Comme pour les définitions de contexte, il n'y a pas de réel consensus autour de la définition de SOA. L'OASIS (Organization for the Advancement of Structured Information Standards) en donne la définition suivante :

Définition 1 : Architecture orientée service

« *Service Oriented Architecture is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations.*[147] »

Dans une architecture orientée service, un service est déposé dans un registre par un fournisseur. Un consommateur peut consulter ce registre afin de trouver un service répondant à ses besoins. Une fois le service trouvé, le consommateur, à l'aide de la description du service et à la suite d'une négociation, peut alors intéragir avec ce dernier. L'objectif de cette approches est de favoriser la réutilisation de services par différent consommateurs. Les services sont les unités de base et sont composées de manière externalisée dans une approches « *business-driven* ». Le couplage entre les services est faible et les dépendances entre eux minimales.

Un fournisseur ne peut pas présumer de la façon dont sera utilisé son service. Fournisseurs et consommateurs ont de grandes chances d'utiliser des architectures matérielles et logicielles différentes. Les services web [153] proposent d'apporter une interopérabilité au niveau des protocoles de communication afin de répondre à cette problématique. Les SOA peuvent être réalisées à partir de services web, on parle alors de *web service oriented architecture* (WSOA) [156]. Les WSOA sont

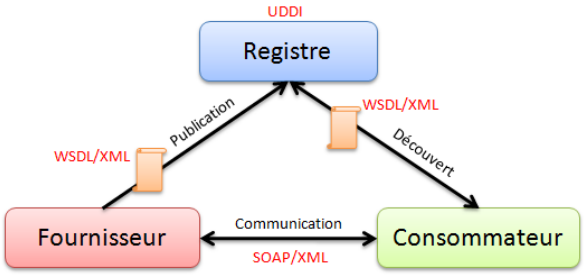


FIGURE 2.5 – Architecture orientée service web.

désormais très largement utilisées dans l'industrie. Les services web, pour fournir cette interopérabilité, reposent sur des technologies tirées du web et l'utilisation intensive de XML et du protocole HTTP. Ces technologies en partie standardisées [1] par la W3C sont au nombre de quatre : SOAP, XSD, WSDL et UDDI. SOAP (Simple Object Access Protocol) est un protocole minimal pour échanger des messages entre services. Afin de pouvoir être aisément porté sur diverses plates-formes avec diverses technologies, il repose sur l'utilisation conjointe de XML et HTTP. WSDL (Web Service Description Language) permet de décrire le contrat d'un service dans une grammaire XML. UDDI (Universal Description Discovery and Integration) est une spécification qui décrit comment publier et découvrir des services Web sur un réseau.

La composition de services peut se faire avec deux approches : les chorégraphies ou les orchestrations [154]. Les chorégraphies proposent une approche décentralisée dans laquelle il n'y a pas de coordinateur central. Elles supposent que les services sont capables de s'organiser pour communiquer les uns avec les autres ; cela nécessite de les adapter pour gérer leurs interactions. En informatique ambiante, où les services sont souvent embarqués sur des appareils mobiles, et sont vus comme des boites noires, cette hypothèse ne doit pas être considérée comme acquise. L'orchestration repose sur une entité centralisée effectuant tous les appels de méthodes sur les services de l'application et relayant les messages entre les services. Les services qui composent une orchestration n'ont pas conscience d'en faire parti. Généralement les orchestrations de services sont réalisées à l'aide de langages basés sur XML comme BPEL4WS. Une autre approche plus dynamique peut consister à orchestrer les services à l'aide d'assemblages de composants.

2.2.2.2 Programmation par composants

La programmation orientée composant est née d'une évolution de la programmation orientée objet. Cette technologie est désormais très largement utilisée dans l'industrie. Elle aide à la réduction des coûts de fabrication de logiciels en permettant de capitaliser du code en réutilisant des composants déjà définis. On parle de composants sur étagère (Components Of The Shelf - COTS). Szyperski dans [161] défini un composant comme suit :

> **Définition 2 : Composant**
> « *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party.* »

1. UDDI n'est pas un standard W3C

Hormis la granularité des entités logicielles définies, une des principales distinctions entre composants et objets réside dans la spécification d'interfaces bien définies. Un type de composant, une entité logicielle abstraite, se caractérise donc par des interfaces. Elles peuvent être de deux types : (1) requises (sorties) ou (2) fournies (entrées). Les interfaces requises permettent de définir les dépendances entre composants. Les interfaces fournies définissent les capacités offertes par un composant. La présence d'interface requise est une autre évolution par rapport au modèle objet. Dans les objets, les dépendances se font via des références se trouvant dans leur code tandis que l'interface requise permet d'expliciter ces dépendances. Cela permet de réduire l'intrication des classes et permet d'obtenir des entités logicielles avec un couplage faible. D'autre part, cela facilite la gestion des interactions entre composants. Ces interactions prennent la plupart du temps la forme d'appels de méthode ou d'événements entre leurs ports.

Une application prend donc la forme d'un ensemble d'instances de composants appelé assemblage de composants. Une instance de composant est créée par une fabrique, soit au déploiement, soit à l'exécution (*late binding*). Lorsque les plateformes à composants autorisent l'instanciation dynamique de composants, elles offrent alors un ensemble de services comme ajouter, retirer, échanger des composants ou encore connecter deux composants.

Les composants sont généralement vus comme des boites noires dont seules les interfaces requises et fournies sont connues. Ils sont alors considérés comme des boîtes réutilisables et composables mais que l'on ne souhaite pas modifier. Aussi, les composants peuvent être considérés comme des boites blanches dont les détails de l'implémentation sont complètement disponibles. Une troisième possibilité consiste à les imaginer comme des boites grises, c'est-à-dire un compromis entre boîte blanche et noire, si seulement une partie de l'implémentation est visible. Cette vision est utilisée dans les divers modèles de composants comme Fractal [61], SCA [56] ou encore SLCA [87] qui autorisent la création de hiérarchies de composants et donc de composants composites reposant sur des assemblages de composants internes.

Les composants sont plus facilement manipulables que les services. Ils sont créés dans des containers qui peuvent fournir des propriétés non-fonctionnelles comme des services pour reconfigurer des assemblages de composants ou gérer leur cycle de vie. Les composants s'exécutent donc dans un environnement d'exécution totalement contrôlé, et peuvent être instanciés et manipulés à la guise du programmeur, alors que les services sont fixes et subis et que leurs orchestration à base de langage de type XML n'offrent pas les mêmes avantages en terme de dynamicité. Dans les approches à composants, le couplage faible entre composants et le contrôle de leur environnement d'exécution facilite leur remplacement dynamique et en font de bons candidats à l'adaptation compositionnelle [12, 131]. Les services sont plus adaptées à la gestion de l'hétérogénéité et à la distribution des entités logicielles. Des approches multi-paradigmes qui utilisent à la fois des architectures orientées services et des composants ont émergées. Les composants peuvent alors être utilisés pour créer de nouveaux services ou afin de manipuler des orchestrations de services et de créer des services composites comme cela est par exemple proposé par SCA [56] ou SLCA [87].

Dans notre cadre de l'informatique ambiante, nous avons vu qu'en raison des trois axes d'imprévisibilité évoqués en introduction, il est nécessaire de disposer d'un mécanisme d'adaptation compositionnelle. En effet, ce dernier offre l'extensibilité nécessaire pour s'adapter à des situations qui n'avaient pas été prévues à l'avance. Toutefois, il reste envisageable de combiner adaptation paramétrée et compositionnelle afin de tirer bénéfice des deux approches. Pour faciliter la mise en place de ces adaptations, nos applications reposeront sur une plateforme d'exécution à base de composants. La modularité, le couplage faible entre les composants et leur dynamicité en font de bons candidats à l'adaptation compositionnelle.

L'adaptation compositionnelle nécessite, lors de sa mise en œuvre, d'interrompre la sous-partie de l'application modifiée. Nous souhaitons limiter le nombre de ces interruptions. D'autre part, avant de porter ces modifications dans l'application, il est important de vérifier leur bonne cohérence pour ne pas être obligé, a posteriori, de corriger par une nouvelle adaptation le résultat d'une ou plusieurs modifications. Nous allons maintenant nous intéresser aux techniques que nous pouvons trouver dans la littérature pour répondre à cette problématique.

2.3 Adaptation et projection sur des représentations abstraites

Une adaptation peut se composer d'un ensemble d'instructions atomiques de modifications d'une application (par exemple l'instruction « ajouter un composant »). Traditionnellement, chaque modification, lorsqu'elle est mise en œuvre, est directement portée dans l'application de manière implicite. L'application est alors interrompue lors de la réalisation successive de chaque modification. Nous verrons, dans la première partie de cette section que cette approche est adoptée par la plupart des plateformes d'exécution adaptatives à base de composants. Après avoir discuté des limitations que cela impose en terme d'interruptions de l'application, nous verrons dans la suite de cette section comment les approches utilisant des modèles de l'application à l'exécution permettent de répondre à ces problématiques.

2.3.1 Quelques plateformes adaptatives à base de composants

De nombreuses plateformes d'exécution adaptatives à base de composants existent et permettent de réaliser des adaptations compositionnelles. B. Morin dans sa thèse [96], montre que la plupart de ces plateformes d'exécution fournissent des API pour l'introspection et la reconfiguration. Ces deux techniques sont les principes de base de la réflexivité. La réflexivité est la capacité d'un système à observer et altérer son comportement [148]. Elle ne s'applique pas uniquement à la programmation orientée objet mais, avec le développement des machines virtuelles, son utilisation s'est désormais largement répandue. Elle repose sur deux mécanismes : (1) l'intercession, qui est la capacité d'un système à modifier son comportement et (2) l'introspection qui est la capacité d'un système à s'observer. Pour ce faire, la réflexion repose sur deux niveaux. L'introspection consiste en particulier en la réification des entités logicielles composant l'application à un niveau méta. L'intercession, quant à elle, est le mécanisme qui permet de modifier ces entités réifiées. Le niveau méta, qui englobe les réifications et le code pour modifier ces entités est en connexion causale avec le niveau de *base* (l'application s'exécutant). Les modifications au niveau *méta* sont directement reportées au niveau de base de manière implicite, c'est le mécanisme d'absorption. Le niveau méta peut être vu comme un modèle de l'application s'exécutant [125]. De la même manière, dans les plateformes d'exécution que nous allons présenter, les modifications sont directement reportées au niveau de base. Hormis le modèle à

composant Fractal, les modèles OSGi, SLCA et SCA mélangent des concepts tirés à la fois des approches orientées composants et services. A chaque modèle nous associerons une implémentation de référence qui offre des mécanismes d'adaptations compositionnelle.

Fractal est un modèle à composant [61] dont une implémentation de référence est Julia [2]. Dans ce modèle, l'entité de base d'une application est un composant. Il peut être primitif ou composite. Un composant primitif est une entité atomique dépendant d'une implémentation du modèle. Un composant composite se compose de plusieurs composants, il englobe un assemblage de composants. Le modèle de composant Fractal est donc hiérarchique ; des composants peuvent englober d'autres composants de manière récursive. De plus, les composants peuvent être partagés, c'est-à-dire présents dans l'assemblage de plusieurs composites. Un composant Fractal se divise en deux parties : un *contenu* et une *membrane*. La membrane d'un composant peut contenir des intercepteurs ainsi que des contrôleurs. Les intercepteurs ont pour objectif d'intercepter les appels à certaines méthodes et les redirigent vers des contrôleurs. Les contrôleurs ont pour objectif de définir de quelle manière contrôler le contenu d'un composant. Les contrôleurs permettent alors par exemple de gérer l'assemblage interne d'un composite. Fractal fournit une API réflexive permettant de créer, introspecter et gérer composants, interfaces ou encore liaisons.

SCA pour Service Component Architecture est un standard avancé par l'OSOA [3] qui propose d'agréger des concepts tirés des approches orientées composants et services [56]. Parmi les implémentations de référence, nous trouvons FraSCAti [111], une implémentation du standard au-dessus de Fractal. Il s'agit de réaliser des compositions de services par assemblage de composants. La réalisation d'un assemblage de composants permet de créer un service composite. L'élément de base d'une composition est le composant. Un type de composant peut avoir plusieurs implémentations à partir du moment où elles respectent sa description. Une description contient les références aux services utilisés par le composant ainsi qu'une description des services et des propriétés qu'il propose. Les fonctionnalités exposées par un composant prennent la forme de service. Le standard ne fait aucune proposition quant à l'adaptation dynamique d'assemblages. Cependant, FraSCAti propose de tels mécanismes grâce à la mise en œuvre du standard au dessus de Fractal.

OSGi est un consortium composé de nombreux grands acteurs industriels [4]. OSGi propose un modèle de composant orienté services, c'est-à-dire que des services sont implémentés à l'aide de composants [93] dont Felix [5] est une implémentation de référence. Les services sont déployés dans des « *bundles* » (composants). Un *bundle* peut être un demandeur ou un fournisseur de services. Physiquement, un *bundle* correspond à un fichier JAR contenant du code et des ressources. Les *bundles* OSGi gèrent eux-mêmes leurs dépendances. De cette manière, un *bundle* peut chercher lui-même des services requis satisfaisant ses dépendances. Un *bundle* peut également s'abonner à un mécanisme de notification l'informant quand un service est disponible. Le framework OSGi, qui est l'environnement d'exécution des *bundles*, fournit un ensemble de mécanismes pour gérer leur cycle de vie. L'administration, locale ou distante d'un framework permet l'installation, le démarrage, l'arrêt ou encore la mise à jour des *bundles*. Cependant, la gestion des dépendances interne aux *bundles* reste complexe.

2. http://fractal.ow2.org/
3. http://www.osoa.org/display/Main/Home
4. http://www.osgi.org/Main/HomePage
5. http://felix.apache.org

SLCA pour Service Lightweight Component Architecture est un modèle multi-paradigmes utilisant : services, composants légers et événements [87]. WComp [6] en est l'implémentation de référence. Comme SCA, ce modèle permet de réaliser des compositions de services par assemblage de composants. Pour ce faire, des composants proxy vers des services Web ou des services pour dispositifs peuvent être générés dynamiquement et intégrés dans un assemblage. Un assemblage de composant se trouve dans un conteneur qui peut lui-même être exporté comme un service composite. Le modèle est donc hiérarchique puisque ce service pourra être utilisé dans un autre service composite. Les communications entre les composants sont événementielles. Un service composite exporte deux interfaces : (1) l'interface structurelle, qui permet de gérer le contenu du composite (ajouter/retirer des composants/liaisons ou encore modifier des propriétés) et (2) l'interface fonctionnelle qui permet de communiquer avec les composants internes du composite.

2.3.1.1 Discussion

Ces plateformes permettent de réaliser à l'exécution des adaptations compositionnelles. Comme nous l'avons vu, dans le cadre de l'informatique ambiante, le système sera appelé à être adapté de nombreuses fois. Dans ces plateformes, lorsqu'une modification est réalisée elle est directement reportée dans l'application, et ce, de manière implicite. Ce qui signifie que la moindre modification atomique est directement transposée de manière à ce que le niveau méta soit toujours l'exact reflet du niveau de base. Il n'y a pas de séparation claire entre les entités méta manipulées et les entités de l'application s'exécutant. Ainsi, comme cela est mis en avant dans [96] : « *Modifying the model implies modifying the reality : there is no mean to preview the effect of a reconfiguration before actually executing it, or to execute what-if scenarios to evaluate different possible configurations, etc.* » . Puisque nous souhaitons mettre en œuvre diverses adaptations que nous ne connaissons pas nécessairement à l'avance, ce type d'approche consisterait à réaliser les adaptations les unes après les autres puis à analyser le résultat déjà porté sur l'application s'exécutant afin de résoudre de potentiels conflits. Reporter les modifications directement dans l'application peut s'avérer risqué [54] et ne permet ni de juger de l'impact de ces modifications sur le système ni de vérifier leur cohérence. Par exemple, si l'ordre dans lequel les modifications sont portées n'est pas soigneusement étudié, peuvent alors apparaître dans l'assemblage des liaisons ouvertes ou encore des cycles qui ne pourront être identifiés qu'une fois l'adaptation réalisée. Il est également possible qu'une modification annule l'effet d'une autre modification. De plus, ce type d'approche peut mener au blocage successif de l'application lors de chaque modification apportée à l'application comme décrit en FIGURE 2.6.

Comme nous l'avons vu dans la section 2.1, nous souhaitons stopper l'application le moins possible durant une adaptation afin de la rendre la plus utilisable possible. Il serait alors intéressant de reporter ces modifications lors d'une unique opération atomique d'adaptation, n'interrompant ainsi l'application qu'une fois. Cela passe, entre autre, par le fait de ne porter dans l'application que les modifications dont la cohérence est vérifiée. L'utilisation unique des plateformes d'exécution adaptative n'est donc pas suffisante. Elles doivent être combinées à un mécanisme permettant de porter « à blanc » les modifications sur une image de l'application puis à déclencher explicitement la mise en œuvre des modifications afin de porter les modifications de manière atomique.

6. http://www.wcomp.fr

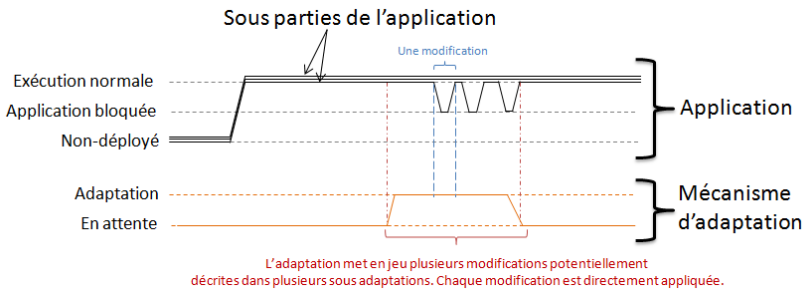


FIGURE 2.6 – Dans les plateformes d'exécutions adaptatives classiques, les modifications portées au niveau méta sont directement transposées dans le niveau de base plutôt que d'être portée une fois que tout le calcul d'adaptation est réalisé.

2.3.2 Appliquer les adaptations sur des représentations abstraites des plateformes d'exécution

Trouvant leurs origines dans le domaine des bases de données, les transactions sont un moyen de réaliser une « unité atomique d'accès à une ressource » donc de porter un ensemble de modifications de manière atomique [141]. Les étapes dans le cycle de vie d'une transaction sont : (1) début, (2) définition de l'ensemble des opérations (3) validation ou (4) abandon puis (5) fin. Aucune modification n'est définitivement portée avant validation. Appliquées à l'adaptation, elles permettraient de réaliser un ensemble d'actions de modifications afin de les appliquer de manière atomique sur le système. Par exemple, le langage FScript [72] permet de garantir les propriétés ACID (atomique, cohérente, isolée et durable) des transactions sur les adaptations.

Cependant, cette approche ne permet pas, avant validation, de vérifier les conséquences de l'ensemble des adaptations. Pour cela, il faut une image de l'application sur laquelle il est possible de réaliser des traitements, sans que cela porte à conséquence. Les travaux dans le domaine des *models@runtime* [125], en proposant de manipuler des représentations abstraites d'une application à l'exécution, offrent un telle possibilité. Si les recherches en ingénierie des modèles portaient au départ sur les phases du cycle de vie d'une application se déroulant avant l'exécution, elles ont évoluées vers l'utilisation de modèles à l'exécution. Couplés à des plateformes d'exécution adaptatives, les *models@runtime* peuvent offrir des capacités d'adaptation compositionnelle dynamique tout en limitant les répercussions apportées aux applications.

2.3.2.1 Une brève introduction à l'ingénierie dirigée par les modèles

L'ingénierie dirigée par les modèles (IDM) regroupe un ensemble d'approches pour le développement de logiciels. Dans ces approches, les entités de bases sont les modèles. Comme pour le contexte, il n'y a pas de définition consensuelle de ce qu'est un modèle. Parmi celles que nous retrouvons dans [129], nous pouvons extraire une des plus ancienne :

Définition 3 : Modèle

« A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system. [124] »

Un méta-modèle est un modèle de modèles. Un modèle est conforme à un méta-modèle s'il vérifie l'ensemble des contraintes exprimées par ce méta-modèle. De la même manière, nous retrouvons des méta-méta-modèle. Un modèle ou un méta-modèle peut être décrit suivant plusieurs formalismes. Dans l'industrie, UML, Meta Object Facility (MOF) ou le Eclipse Modeling Framework (EMF) sont très largement utilisés. Il existe également de nombreux outils pour vérifier la conformité d'un modèle avec son méta-modèle.

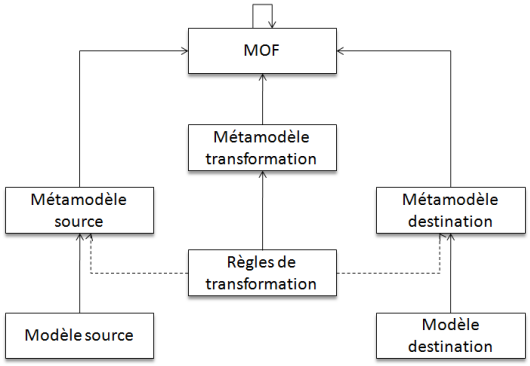


FIGURE 2.7 – Transformation de modèle

Une transformation de modèle est une opération qui prend pour entrée un modèle et produit un modèle (FIGURE 2.7). Une transformation est dite endogène [126], si elle prend en entrée et produit des modèles qui sont conformes au même méta-modèle. A l'inverse, une transformation est dite exogène [126], si elle prend en entrée et produit des modèles qui ne sont pas conformes au même méta-modèle. Ce type de transformation permet par exemple, à partir d'un modèle métier indépendant de toute plateforme (on parle de PIM - Plateform Independent Model) de produire plusieurs modèles propres à diverses plateformes (on parle de PSM - Plateform Specific Model). Une transformation définit un ensemble de règles établissant des correspondances entre éléments des modèles sources et cibles. Des outils comme Kermeta, ATL ou encore XSLT permettent de réaliser des transformations de modèle.

L'ingénierie dirigée par les modèles offre donc divers avantages. Le premier d'entre eux est le niveau d'abstraction fournit par les modèles. Leur utilisation permet de décrire le système au niveau d'abstraction souhaité pour un but particulier. Dans les systèmes complexes, il est généralement admis qu'augmenter le niveau d'abstraction permet de faciliter la gestion du système. Pour l'adaptation, comme cela est décrit par Oreizy, travailler avec un haut niveau d'abstraction facilite le raisonnement nécessaire au calcul de l'adaptation et réduit la quantité d'informations à gérer : « *the level of abstraction at which changes are described impacts the complexity and quantity of information that must be effectively managed.* » [152]. D'autre part, l'utilisation des architectures dirigées par les modèles (MDA) permet d'obtenir un bon découplage entre code métier et plateformes d'implémentation. Un même modèle peut alors être réutilisé dans plusieurs applications sur diverses plateformes. Enfin, les propriétés de conformité et le « *model checking* » offrent des mécanismes automatiques permettant de vérifier la consistance d'une application.

La communauté *models@runtime* propose d'utiliser des modèles pour représenter les applications à l'exécution, et dans l'optique de réaliser des traitements sur ces derniers.

2.3.2.2 Les models@runtime

Les *models@runtime* proposent de tirer bénéfice des approches à modèles à l'exécution. Un modèle représente l'application et peut être manipulé à l'exécution. Les traitements réalisés sur ces modèles peuvent être du monitoring comme de l'adaptation. Dans le cadre de l'adaptation dynamique, les *models@runtime* fournissent un moyen de mesurer l'importance des modifications apportées au système et de les analyser avant de les reporter sur le système s'exécutant [54, 80]. Modifier directement le système, dans certaines situations, peut s'avérer risqué ou menacer d'en détourner l'utilisateur si le bon fonctionnement du système est sujet à de trop grandes perturbations générées successivement. Par exemple, cela peut intervenir lorsque les adaptations sont conflictuelles et qu'il est nécessaire de réadapter le système pour corriger ces conflits. Le modèle peut être en connexion causale avec l'application s'exécutant. Lorsque des modifications sont portées sur le modèle, elles peuvent être reportées sur l'application suivant plusieurs politiques. Plusieurs types de connexions causales existent [54] :

- La connexion peut être forte, de manière à être sûr que le modèle est à jour et reflète bien l'application. Cette approche est semblable à celle généralement proposée par les protocoles à méta-objets [143] qui définissent la manière dont les objets et méta-objets (réification des objets à un niveau méta) interagissent.
- La connexion peut être plus lâche afin de permettre des raisonnements avant de porter les modifications. Dans ce cas, les modifications sur le modèle ne sont pas immédiatement reportées sur l'application. Afin de garantir que les calculs d'adaptation seront bien réalisés sur un modèle reflétant correctement l'application, la synchronisation de l'application vers le modèle à l'exécution est forte. Par contre, la synchronisation du modèle vers l'application s'exécutant est retardée pour permettre ces vérifications. Cette approche est par exemple celle utilisée dans [97, 54]. La principale différence entre les deux approches réside dans le fait qu'ici l'absorption est retardée et peut être déclenchée explicitement après validation.

Comme nous l'avons vu, l'informatique ambiante nécessite l'utilisation de la connexion la plus lâche. En effet, avant de reporter les modifications, nous souhaitons, a minima, réaliser des traitements permettant d'en vérifier la cohérence. Les modèles mis en jeu peuvent avoir plusieurs granularités, selon la représentation choisie et les capacités d'introspection de la plateforme sous-jacente ; il est possible d'avoir plus ou moins d'informations sur les entités qui composent l'application. Si nous abstrayons trop, il est possible de perdre des informations importantes qui pourraient influencer les traitements que nous souhaitons réaliser. Par contre, il est important de noter que plus nous travaillons à un niveau de détail important, plus la synchronisation entre le modèle à l'exécution et l'application s'exécutant peut être longue et coûteuse. Ce point est plus particulièrement important lorsqu'il met en jeu des systèmes complexes dont le nombre d'éléments à inspecter peut être élevé. Ceci intervient dans les deux sens de la synchronisation.

Dans le sens application → modèle : la synchronisation peut être coûteuse en temps lorsque nous utilisons uniquement les capacités d'introspection du système pour observer l'ensemble du système afin de créer ou de maintenir à jour le modèle de l'application s'exécutant. Pour limiter ce phénomène, il est possible de ne pas mettre à jour tout le système mais, par exemple, de faire évoluer en parallèle les deux niveaux (méta et de base) pour ne mettre à jour que ce qui est modifié. Il faut alors que la plateforme d'exécution propose un mécanisme de notification de manière à ce que chaque modification dans l'application soit à l'origine d'une notification permettant de mettre à jour le modèle.

Dans le sens modèle → application : pour éviter que la synchronisation soit trop coûteuse en temps, il n'est pas nécessaire de reconstruire entièrement l'application. Il est possible, par exemple, de comparer modèle et application dans une vision ensembliste [96]. Les éléments présents dans

le modèle adapté et dans l'application source sont conservés, les éléments n'étant pas dans le modèle mais présents dans l'application sont retirés, tandis que les éléments du modèles n'étant pas présents dans l'application sont ajoutés. Il s'agit donc de modifier uniquement ce qui a changé dans l'application.

Ce type d'approche est donc utilisé dans diverses plateformes d'adaptation. Par exemple, Bencomo *et al* avec Genie, l'utilisent afin de générer et de reconfigurer des systèmes à base de composants [54, 55]. La plateforme d'exécution sous-jacente a pour objectif de mettre en œuvre les reconfigurations décrites dans des scripts. Pour ce faire, deux types de modèles sont définis. Le premier permet de spécifier des reconfigurations tandis que le second décrit l'ensemble des transitions permettant de passer d'une configuration à une autre. Toutes les transitions sont donc spécifiées explicitement. Une fois qu'une configuration spécifiée est identifiée comme pertinente dans la situation courante, elle est comparée à un modèle du système s'exécutant. Les différences sont alors à l'origine de scripts de reconfigurations à effectuer sur l'application. Les traitements sont réalisés au niveau méta avant de reporter les modifications au niveau de base (FIGURE 2.8).

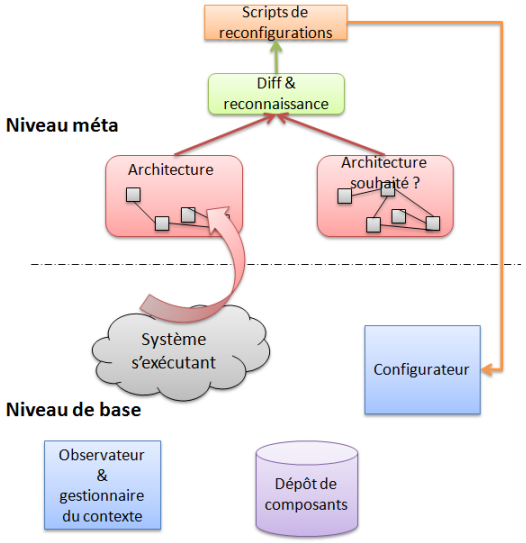


FIGURE 2.8 – Niveaux de base et niveaux méta dans l'approche proposée par Bencomo *et al* [54].

Il est également important de noter que lorsque l'on utilise des modèles d'application à l'exécution, modèles et applications ne sont pas nécessairement en connexion causale. C'est par exemple le cas lorsque les modèles sont utilisés uniquement afin de faire du monitoring. Dans le projet MADAM [76], un architecte spécifie un modèle d'architecture qui peut être instancié de diverses manières définissant ainsi différentes variantes de l'application conformes au modèle d'architecture. Ces différentes variantes sont définies en faisant fluctuer les différents composants qui peuvent être utilisés pour les réaliser. Des fonctions d'évaluation ont pour objectif d'évaluer les modèles des différentes variantes en fonction du contexte afin de choisir la plus pertinente. L'application s'exécutant est elle aussi représentée à l'exécution sous la forme d'un de ces modèles afin d'être évaluée avec les autres. Une fois la nouvelle configuration sélectionnée, le composant de reconfiguration compare cette dernière avec l'application afin de reporter les modifications. Dans ce cas, le modèle de l'application n'est pas maintenu en relation causale avec elle mais est simplement utilisé lors de la phase d'évaluation des différentes configurations.

Le framework Rainbow [81, 80] utilise également un modèle abstrait architectural représentant l'application s'exécutant. Ce modèle prend la forme d'un graphe dont les nœuds représentent les principaux éléments logiciels de l'application. Les arcs incarnent les interactions entre ces composants logiciels. A chacun de ces éléments peuvent être associés des propriétés. Une application s'exécutant est observée à l'aide d'un mécanisme de monitoring. Ces observations permettent de construire et d'associer des propriétés au modèle architectural. Lorsque des propriétés évoluent, un ensemble de contraintes associées au modèle sont évaluées. Si des contraintes sont violées, l'architecture est adaptée et les changements sont propagés jusqu'à l'application s'exécutant.

Si pour évaluer les effets sur l'application d'un ensemble de modifications il est important de disposer d'un modèle représentant fidèlement l'application, la modification de l'application n'a pas nécessairement besoin d'être liée à la modification du modèle. La mécanisme d'introspection joue donc un rôle particulièrement important et, comme nous l'avons vu, son utilisation peut être limitée si le modèle de l'application et l'application sont modifiés de concert (que la connexion causale soit lâche ou non). Comme nous l'avons vu, cela permet de conserver un modèle qui évolue avec l'application ce qui ne nécessite pas d'introspecter à nouveau l'application. En conséquent, nous privilégierons ce type d'approche.

Nous nous intéresserons donc aux adaptations réalisées sur des représentations abstraites des applications sur lesquelles elles portent. Ceci permet de reporter les modifications dans l'application seulement lorsque l'ensemble de celles-ci ont étés calculées et vérifiées. De cette manière, un mécanisme de résolution de conflits entre adaptations peut intervenir avant la modification de l'application. Ceci est particulièrement important lorsque nous nous trouvons dans un cadre pour lequel l'ensemble des adaptations ne peut pas nécessairement être anticipé. D'autre part, cela permet de réaliser les adaptations sur des représentations abstraites pouvant fournir un plus haut niveau d'abstraction que celui de l'application, ce qui peut réduire la complexité des calculs d'adaptations. Nous privilégierons les modèles en connexion causale lâche avec l'application qu'ils représentent. Nous souhaitons réaliser des traitements, au minimum de vérification sur la cohérence des modifications à apporter à l'application, avant de les reporter sur l'application s'exécutant. Toutefois, tout ceci nécessite l'utilisation d'un mécanisme de synchronisation entre l'adaptation s'exécutant et sa représentation abstraite. Ce mécanisme peut avoir un coût sur les temps de réponses de l'adaptation même si les modifications peuvent être moins nombreuses et moins fréquentes. Il est donc important de minimiser l'impact de ce mécanisme.

Nous souhaitons donc réaliser des adaptations compositionnelles dynamiques, tout d'abord sur des représentations abstraites des applications, avant de les reporter dans ces dernières. Cependant, l'environnement est trop variable pour que l'on puisse écrire de manière ad-hoc toutes les configurations d'une application ainsi que toutes les adaptations permettant de passer d'une configuration à une autre. Modulariser et factoriser au maximum les adaptations, configurations, afin de pouvoir les réutiliser et les composer le plus aisément possible permet de décrire un maximum de configuration à partir d'un petit nombre d'entités d'adaptation. Cependant, bien souvent, les adaptations impactent plusieurs fonctionnalités d'une application et sont difficilement modularisées et découplées de ces dernières. Le principe de la séparation des préoccupations offre une solution à cette problématique. Dans la section suivante, nous identifierons dans un premier temps ces besoins de réutilisabilité et de

modularité à la suite de quoi nous présenterons le principe de la séparation des préoccupations. Nous verrons, en particulier, la difficulté à mettre en application ce principe lorsque les préoccupations à séparer sont transverses à d'autres. Nous détaillerons ensuite plusieurs techniques permettant de mettre en œuvre la séparation des préoccupations transverses. Enfin nous montrerons que l'adaptation est une préoccupation transverse et qu'il est possible de lui appliquer les techniques que nous aurons étudiées.

2.4 Séparation des préoccupations : découpler les propriétés fonctionnelles des propriétés transverses

La variabilité ainsi que l'envergure des phénomènes auxquels peuvent être sensibles les systèmes ambiants mènent souvent à la mise en place de systèmes complexes proposant de nombreuses fonctionnalités. Il devient alors vite difficile de maîtriser ou de maintenir ces systèmes. En effet, bien souvent, différentes fonctionnalités se trouvent offertes par un même morceau de code. C'est le cas dans les approches consistant à décrire l'ensemble des configurations atteignables ; un même code se trouve souvent disséminé dans plusieurs configurations. Il en va de même lorsque nous écrivons de manière ad-hoc toutes les adaptations. Modifier ce code disséminé implique alors de modifier indépendamment chaque configuration, adaptation.

Nous retrouvons, par exemple, ce type d'approche dans Genie [54], une plateforme pour laquelle nous avons vu qu'il est nécessaire de définir l'ensemble des variantes de configurations du système et les transitions entre elles. C'est également le cas de l'intergiciel CARISMA (Context-Aware Reflective mIddleware System for Mobile Applications). Il s'agit d'un intergiciel réflexif qui peut être personnalisé en fonction des besoins des applications [62]. A chaque application peut être associé un profil. Ces profils définissent les associations entre fonctionnalités fournies, politique de mise en œuvre de cette fonctionnalité et le contexte permettant l'identification d'une fonctionnalité. Ces profils peuvent être modifiés pendant l'exécution de l'application grâce à la notion de réification des profils. Dans cette approche, un seul profil peut être associé à une application et les associations dans divers profils peuvent être dupliquées. Les préoccupations dans un même profil sont entrelacées. L'API permettant la modification d'un profil nécessite l'écriture ad-hoc des modifications de ce dernier.

Ce type d'approche limite la modularité et la réutilisabilité des adaptations ainsi que des diverses fonctionnalités de l'application. L'entrelacement du code des adaptations et des diverses fonctionnalités rend ardue l'identification du code à modifier et ne facilite pas l'intégration de ces modifications. Le manque de réutilisabilité que cela entraîne, implique bien souvent l'écriture ad-hoc de l'ensemble des configurations du système. Or, nous avons vu que l'environnement est trop variable et imprévisible pour que l'on puisse écrire de manière ad-hoc, pour chaque variation de l'environnement, toutes les configurations d'une application. Il n'est pas non plus possible d'écrire toutes les transitions possibles entre ces configurations ou encore toutes les adaptations. Le principe de la séparation des préoccupations [142] propose de répondre à cette problématique. Il s'agit d'un principe de conception empruntant une stratégie de type : « diviser pour mieux régner » qui permet de décomposer et d'encapsuler les différentes fonctionnalités du système dans des entités modulaires.

2.4.1 La notion de séparation des préoccupations

Le principe de la séparation des préoccupations propose de décomposer un système logiciel complexe en diverses préoccupations faiblement couplées que l'on peut gérer de manière isolées. Ainsi, chaque élément du système ne traite qu'une préoccupation et doit être compréhensible par lui-même. Cela aide à structurer le code, facilite sa compréhension ainsi que sa maintenance, et permet lors du développement de se concentrer sur une préoccupation en particulier. Une préoccupation se définit comme suit :

Définition 4 : Préoccupation

« *A concern is an area of interest or focus in a system. Concerns are the primary criteria for decomposing software into smaller, more manageable and comprehensible parts that have meaning to a software engineer.* » [1]

Les concepteurs décomposent alors leurs systèmes en fonction de préoccupations qu'ils considèrent comme principales. Diverses techniques peuvent être utilisées pour réaliser ce type de conception modulaire comme les approches orientées objets, composants ou même procédurales (une préoccupation équivaut à une procédure). Cette décomposition augmente la réutilisabilité du code et offre un premier moyen de gérer la variabilité d'un système. Cependant, il subsiste, entrelacé avec les préoccupations principales, le code d'autres préoccupations qui n'ont pas pu être modularisées comme les autres. On qualifie ces préoccupations de « transverses » [7] (FIGURE 2.9), c'est-à-dire qu'elles sont transverses à plusieurs préoccupations de base. Cet enchevêtrement de code rend sa maintenance, sa compréhension et ses mises à jour difficiles. Par exemple, modifier le code d'une préoccupation transverse peut avoir des répercutions sur le code de plusieurs préoccupations principales. Bien souvent, les préoccupations transverses sont des préoccupations non fonctionnelles comme la gestion des transactions ou encore de logs.

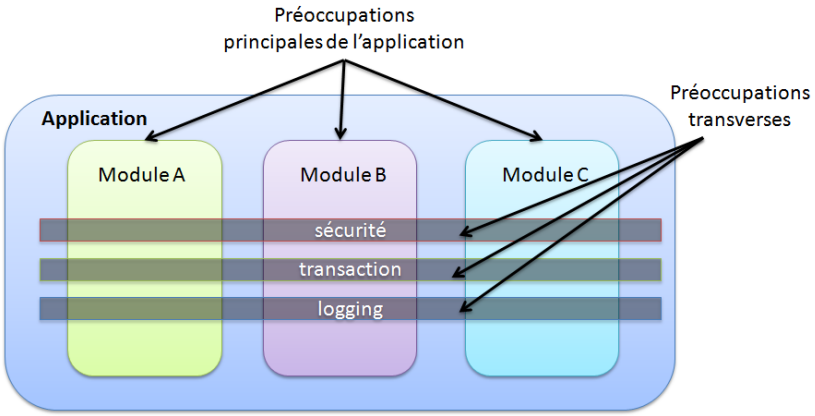


FIGURE 2.9 – Préoccupations coupant de manière transversales plusieurs modules.

Deux grands types de préoccupations transverses ont étés identifiées dans la littérature. Une préoccupation transverse est dite « homogène » si sa mise en œuvre consiste en la duplication d'un même code en plusieurs endroits du système. Inversement, une préoccupation est dite « hétérogène » si sa mise en œuvre implique des morceaux de codes différents dans divers endroits du système [4]. D'autres approches que les paradigmes objets, composant ou encore services sont alors proposées

afin de permettre la mise en œuvre du principe de séparation des préoccupations sur ces préoccupations transverses. Ces approches sont complémentaires aux paradigmes précédemment évoqués. Parmi celles-ci, deux ont pris une place prépondérante : (1) la programmation orientée aspect (AOP) plus particulièrement adaptée à la gestion des préoccupations transverses homogènes [3] et (2) la programmation orientée feature (FOP) plus particulièrement adaptée à la gestion des préoccupations transverses hétérogènes [3]. Cependant, la FOP, ne permet pas comme l'AOP, d'exprimer tous les types de préoccupations transverses [4]. Nous allons maintenant détailler ces deux approches.

2.4.2 La Programmation Orientée Aspects (AOP)

La programmation par aspects (AOP) a été proposée par Kiczales *et al* en 1997 [15, 17]. Il s'agit d'un paradigme qui permet de définir des abstractions logicielles qui viendront se superposer à une application de base. A ses origines, la programmation par aspects est partie du constat suivant : malgré tous les efforts faits en ce sens, il existe toujours un fort entrelacement entre le code métier d'une application et celui des préoccupations transverses (sécurité, monitoring ...). L'idée de la programmation par aspects est donc de représenter de manière séparée, dans des aspects, ces préoccupations transverses. Il s'agit alors d'injecter le code des préoccupations transverses dans le code de base. Cette injection se fait par l'intermédiaire d'un *tisseur* d'aspects.

Un aspect se découpe en points de coupe et greffons. Les premiers identifient « où » le code doit être tissé tandis que les seconds décrivent le code à injecter, le « quoi » (Fig. 2.10). L'ensemble des attaches sur lesquelles peuvent être tissées des aspects s'appellent des points de jonctions. Selon les paradigmes sur lesquels repose la programmation par aspects, cet ensemble change de nature (objets, composants, code ...) mais l'AOP offre toujours une bonne séparation des préoccupations, une bonne modularité transverse [7]. Les points de coupes permettent donc de choisir un sous-ensemble de points de jonction sur lesquels seront ajoutés un comportement. L'abstraction fournie par les points de coupes permet à un aspect d'être tissé en plusieurs endroits de l'application ; la programmation par aspects permet donc de minimiser la dispersion du code, en le regroupant dans des entités réutilisables. Classiquement, les langages de greffon offrent des mécanismes pour ajouter du comportement aux points de coupe à l'aide des opérateurs *after*, *before* et *around*. Ainsi le greffon *before* sera exécuté avant l'exécution du point de jonction vérifiant le point de coupe qui lui est associé et inversement avec un greffon *after*. Un greffon *around* permet soit de remplacer le point de coupe soit d'exécuter du code avant et après le point de coupe.

```
1 public aspect Nom_Aspect {
  pointcut Nom_Method() : // code

4 //// Greffon
  before():Nom_Method() {
        // code
7 } }
```

FIGURE 2.10 – Modèle d'un aspect en AspectJ

Avec le temps, les aspects ont étés couplés à différents paradigmes parmi lesquels les services [6, 7] ou encore les composants [21] qui, comme nous l'avons vu précédemment, sont d'excellents candidats pour la création d'applications en informatique ambiante. Ce couplage peut être envisagé

avec deux visions différentes. Cela peut consister à modifier un composant avec une approche invasive qui entraîne la perte des propriétés de boite noire des composants. Sinon, cela peut consister en des modifications de la structure d'un assemblage comme le propose par exemple FAC (Fractal Aspect Component) [101] qui intègre la notion d'aspects dans Fractal [61]. Il propose alors la définition de *Aspect Component (AC)*, c'est-à-dire de composants encapsulant une préoccupation transverse. Le code du composant représente alors le greffon de l'aspect et les points de coupe sont définis au niveau du connecteur. Les points de jonction sont les composants, les interfaces et les méthodes sur lesquels un AC peut être appliqué. L'application d'un AC peut se faire à l'exécution. Navase *et al* [19] mettent alors en avant les avantages suivants pour le couplage entre une approche orientée composant avec des aspects non invasif : faciliter la conception de l'architecture de l'application, réduire le coût de développement, augmenter la réutilisation de section d'architecture et réduire le coût de la maintenance.

Le tisseur est le mécanisme qui prend en entrée ces aspects ainsi qu'une application et produit une application augmentée. Les premiers mécanismes de tissage comme dans AspectJ [14] étaient statiques et intervenaient à la compilation. Avec AspectJ, le code décrit dans les greffons est tissé dans le code de l'application pour générer un nouveau fichier, par exemple *.java* ou *bytecode*. Ainsi, la séparation des préoccupations introduite par les aspects n'a plus lieu à l'exécution. Par la suite, des approches permettant un tissage dynamique sont introduites. Dans la plupart des approches, les aspects sont alors tissés séquentiellement même si quelques approches proposent de gérer la concurrence entre aspects comme par exemple Douence *et al* [8]. Dans ces approches l'application d'un aspect ne bloque pas nécessairement complètement l'exécution de l'aspect de base jusqu'à ce que le greffon soit totalement tissé. L'approche proposée consiste donc à étendre leurs premiers travaux sur EAOP, pour lesquels les aspects étaient tissés en séquence, avec un tissage concurrent (CEAOP). Pour cela des opérateurs de synchronisation sont intégrés. Si les aspects s'appliquant sur un même point de jonction peuvent toujours être appliqués en séquence, des opérateurs (ParOr, ParAnd) permettent désormais leur application en parallèle.

Les aspects ne sont pas toujours indépendants les uns des autres ; des problèmes d'*interactions* peuvent apparaître entre eux. Une interaction entre deux aspects se défini comme une relation intervenant entre deux aspects, impliquant la perte de leur indépendance [23]. Sanen *et al* [22, 23] ou encore Greenwood *et al* [13] identifient différents types d'interactions parmi lesquelles nous pouvons retrouver, par exemple, la notion de *renforcement* lorsque la présence d'un aspect accentue le bon comportement sémantique d'un autre aspect. Une interaction est qualifiée de *conflit* lorsqu'un aspect requiert l'absence d'un autre aspect pour avoir un comportement sémantiquement correct. Lorsque cette relation entre deux aspects est conflictuelle, c'est-à-dire qu'au moins un des deux aspects ne fonctionne plus comme prévu, on parle d'*interférence*. Sanen *et al* dans [22] définit un interférence comme suit :

Définition 5 : Interférence

« *We define aspect interference as a conflicting situation where one aspect that works correctly in isolation dœs not work correctly anymore when it is composed with other aspects.*[22] » .

Classiquement la gestion des interférences entre aspects se fait en établissant un relation d'ordre entre eux. On parle de précédence [16]. En effet, l'opération de tissage d'aspects n'est généralement pas symétrique $f \otimes g \neq g \otimes f$ [16]. Quelques travaux proposent de s'attaquer à la gestion des

interactions principalement en fournissant des mécanismes de détection d'interférence [2] et en fournissant des outils pour aider à leur résolution [24, 5]. Celle-ci reste généralement à la charge du développeur comme l'approche *step-wise* proposée dans [24]. Il en va de même dans EAOP [9], pour lequel la gestion des interactions peut se faire de deux manières : (1) en contrôlant la visibilité d'aspects par leur encapsulation dans d'autres aspects et (2) en composant explicitement des greffons intervenant sur un même points de jonction (a. séquence, b. sélection d'un unique aspect, c. séquence arbitraire décidée par l'analyseur). Une autre approche présentée dans [2], consiste à simuler et à représenter les différents états au runtime d'un programme sous la forme d'un graphe puis à identifier les interférences comportementales entre les aspects et en particulier en fonction de leur ordre d'exécution.

Grâce à l'abstraction offerte par les points de coupe, un même aspect peut être instancié en plusieurs endroits d'une même application. Les aspects sont donc plus particulièrement adaptés pour exprimer les préoccupations transverses homogènes. Les préoccupations transverses hétérogènes sont quant à elles plus naturellement exprimées à l'aide la programmation orientée feature qui permet d'intégrer dans une application différentes variantes d'une même fonctionnalités.

2.4.3 La Programmation Orientée Feature (FOP)

Le paradigme de conception orienté feature a pour objectif de permettre la conception d'application par composition de plusieurs features. L'approche, incrémentale, dite de « *step-wise refinement* » consiste à venir ajouter petit à petit des features pour construire une application comme un empilement de couche successives. Une feature se définit comme suit :

Définition 6 : Feature

« *A feature is a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option.* » [27]

Cette approche de conception permet de définir une bonne structuration d'un programme et facilite la réutilisabilité des features qui le composent suivant le principe de la séparation des préoccupations. Une feature peut également englober une préoccupation transverse [27]. Par exemple, les features peuvent être transverses à une architecture orientée objet. Cependant certains types de préoccupations transverses peuvent difficilement être encapsulées de cette manière [4]. Ainsi à partir d'un ensemble de features, plusieurs programmes peuvent être générés sous la forme de diverses combinaisons de features comme nous pouvons le voir dans la FIGURE 2.11 tirée de [27]. Le paradigme de conception par features, lors de l'implémentation, n'est pas nécessairement basé sur une approche de programmation orientée feature. Si tel est le cas, une feature devient une entité de première ordre. Un ensemble de features pouvant être composées les unes avec les autres, elles offrent un espace de solutions variables. Les « *feature model* » proposent de modéliser cette variabilité en exprimant selon quelles possibilités les features peuvent être combinées. Un des principes de base du développement logiciel orienté feature est de générer automatiquement un système logiciel (par exemple une ligne de produit) à partir d'une sélection de features, respectant le feature model, par exemple réalisée par un utilisateur.

Puisque l'approche est incrémentale, classiquement, les features sont composées en séquence jusqu'à l'obtention du système final [127]. L'ordre dans lequel sont composées les features est donc particulièrement important comme le montre la notation mathématique proposée dans [29] à savoir :

$f \bullet g \neq g \bullet f$. D'autre part comme les aspects, des features peuvent être en interaction. Apel *et al* [27] en proposent la définition suivante, présentant de forte similitudes avec celle des interactions entre aspects :

Définition 7 : Interaction entre features
« *A feature interaction is a situation in which two or more features exhibit unexpected behavior that does not occur when the features are used in isolation.* » [27].

Comme pour les aspects des travaux s'intéressent à la détection et à la gestion de ces interactions. Cela peut, par exemple, consister en la définition de contraintes ou encore en l'étude des différents ordres de composition des features à l'aide de « *commuting diagrams* » [30].

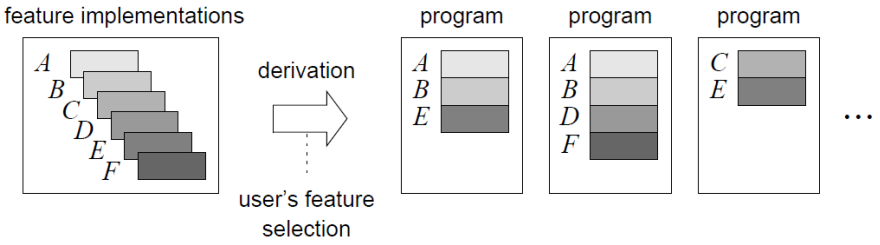


FIGURE 2.11 – Feature oriented programming [27]

Les paradigmes de programmation orientés aspects et features, ne sont pas incompatibles et peuvent être utilisés conjointement afin d'exprimer des préoccupations transverses homogènes et hétérogènes. Plus particulièrement, les aspects peuvent être un moyen d'implémenter des features. Nous allons voir que ces deux approches sont appropriés à l'expression d'adaptations.

2.4.4 L'adaptation comme une préoccupation transverse

David *et al* avec SAFRAN [71] ont montré que la logique d'adaptation pouvait être séparée de la logique applicative. La logique d'adaptation est une préoccupation particulière du système et peut avoir des effets sur plusieurs autres préoccupations. Il s'agit donc bien d'une préoccupation transverse. Cette logique d'adaptation englobe elle même un ensemble de préoccupations d'adaptation liées à des besoins divers. Nous avons également vu que dans les approches transparentes, les mécanismes d'adaptation sont sortis des applications puisque suffisamment génériques. Les entités d'adaptation doivent donc être externes aux applications et gérées comme des préoccupations transverses. En raison de la complexité de ces systèmes et de la multiplicité des dispositifs proposant des fonctionnalités parfois équivalentes, une même adaptation peut être dupliquée en plusieurs endroits de l'application. Il faut donc augmenter la réutilisabilité et la modularité de ces adaptations de manière à ne pas devoir écrire explicitement toutes les configurations du système ou encore d'écrire de manière ad-hoc chaque adaptation permettant d'atteindre ces configurations.

Dans le cadre de l'informatique ambiante, Zambrano *et al* dans [26] expliquent comment l'AOP peut être utilisée pour la création d'applications ubiquitaires. Puisque l'AOP permet d'offrir un bonne modularisation et séparation des préoccupations transverses, elle permettrait, dans le cadre de

l'informatique ambiante, d'isoler des préoccupations variables de l'application. C'est-à-dire qu'une application en informatique ambiante à une base et celle-ci peut être agrémentée ou adaptée par des aspects. Ils permettent alors d'isoler, réutiliser ou encore échanger ces préoccupations en fonction de l'environnement de l'application. La préoccupation de prise en compte du contexte peut alors être séparée du reste de l'application. Comparée à la programmation orientée objet ou composant, il n'y a plus de dépendance entre prise en compte du contexte et logique métier.

De plus l'informatique ambiante met en jeu des systèmes dont les designers doivent considérer les caractéristiques suivantes : mobilité, variabilité des ressources et de leur accessibilité. Tous ces critères, lorsque pris en compte, impactent la manière dont l'application va être mise en place. Ces variabilités ont pour conséquence, dans une approche classique de conception, de forcer la réécriture de même préoccupations pour chaque variante. L'AOP permet de réduire la réécriture de ces préoccupations grâce à l'abstraction offerte par les points de coupe. Les avantages des aspects identifiés sont les suivants :

- la partie adaptation est transparente pour l'application de base,
- les préoccupations en relation avec l'ubiquité du système peuvent évoluer indépendemment les unes des autres,
- la représentation du contexte est conservée coté client,
- cela permet une réutilisabilité des représentations et des stratégies d'adaptation.

La mise en œuvre de la séparation des préoccupations est alors un premier moyen de gérer la variabilité des systèmes ambiants. A l'inverse les inconvénients sont les suivants :

- comment garantir que le comportements de divers aspects ne sont pas conflictuels ?
- le choix des aspects peut introduire des interférences entre applications de base et aspects.

Nous rajoutons à ces inconvénients le suivant :

- les aspects peuvent interférer entre eux.

De nombreuses plateformes proposent d'utiliser des aspects comme moyen de mettre en œuvre des adaptations dynamiques et décrivent les adaptations dans des entités réutilisables et séparées de l'application. Le tableau 2.1 présente quelques unes de ces plateformes. Nous allons maintenant détailler trois d'entre elles.

SAFRAN [71] est une extension de Fractal ayant pour objectif de faciliter la conception de systèmes adaptatifs. Pour ce faire, ils utilisent des *aspects d'adaptation*. Ils peuvent être ajoutés/retirés à l'exécution. Le modèle de point de jonction de SAFRAN ne se limite pas au flot d'exécution de l'application. Les adaptations peuvent être déclenchées par des événements relatifs au contexte de l'application. Les événements *exogènes* sont relatifs aux informations externes à l'application tandis que les événements *endogènes* sont relatifs aux informations internes à l'application. L'architecture de SAFRAN comprend deux parties : (1) un langage d'adaptation appelé FScript pour reconfigurer des assemblages de composants Fractal garantissant les propriétés ACID [72] ; et (2) une boîte à outils pour l'observation du contexte appelée WildCAT [70]. Enfin, un contrôleur est intégré à la membrane afin de lier, à l'aide de règles de type ECA (event-condition-action), ces deux parties et gère les dépendances entre adaptations de manière explicite.

La « *plugin architecture* » proposée dans [6] se base sur AO4BPEL [7] qui est un langage orienté aspects de workflow. AO4BPEL permet l'adaptation dynamique d'orchestration de services. Les approches d'orchestration classiques ont les limitations suivantes [156] : (a) manque de moyens pour

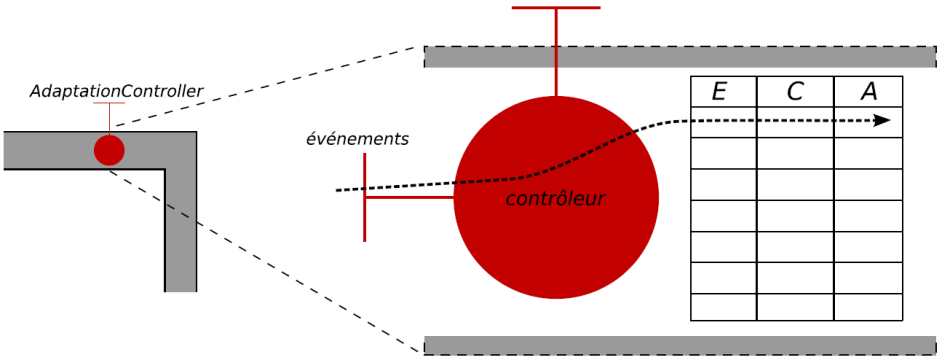


FIGURE 2.12 – Contrôleur d'adaptation dans la membrane d'un composant Fractal [68].

la représentation de préoccupations transverses et (b) pas d'adaptation dynamique de la composition de service. AO4BPEL propose alors un langage plus flexible pour la composition des Web services. Dans ces travaux, la problématique de gestion des interférences entre aspects n'est pas gérée. La gestion se fait de manière classique à l'aide des opérateurs *before, after...*. Puisque ces travaux s'appliquent aux workflows, ils ne prennent pas en compte l'évolution de l'infrastructure logicielle du système. Dans l'architecture, nous pouvons identifier deux types d'aspects : (1) les *monitoring aspects* qui sont capables d'activer ou désactiver des (2) *adaptation aspects* à l'exécution. Les aspects peuvent être ajoutés ou retirés ou parfois générés à l'exécution.

AspectOpenCOM [11] est une extension du modèle à composant OpenCOM [66]. Elle ajoute deux fonctionnalités au modèle. La première est la possibilité de réaliser des compositions par aspects. La seconde est la définition d'un « *aspect-MOP* » qui permet l'introspection des aspects et leur adaptation. Il est alors possible d'observer et de modifier la logique de composition de ces aspects et les comportements qu'ils décrivent. Ce mécanisme offre, par exemple, la possibilité de réordonner les greffons lorsqu'ils sont en interaction. Les aspects prennent la forme de composants. La mise en place de ces aspects consiste en l'interception des invocations sur les ports fournis/requis de composants, et en la réalisation de traitements avant ou après.

Nous voyons dans le TABLEAU 2.1 que la plupart des approches sur les composants permettent d'utiliser des aspects sans violer la propriété de boîte noire des composants de l'application. D'autre part, dans ces approches, la gestion des interférences entres aspects est laissée à la charge d'un développeur ou d'un mécanisme externe.

TABLE 2.1 – Caractéristiques des plateformes d'adaptations à base d'aspects.

Plateforme	**Paradigme associé**	**Forme du greffon**	**invasive/non-invasive**	**Gestion des interférences**
FAC [101]	Composants	Un aspect est un composant	non-invasive	A la charge du développeur
PRISMA [20]	Composants	Un composant est décrit comme une agrégation d'aspects interchangeables dynamiquement	invasive et non-invasive	A la charge du développeur
AO4BPEL [7]	Services	Activité spécifié en BPEL	non-invasive	A la charge du développeur
Aspect-OpenCOM [11]	Composants	Un aspect est un composant	non-invasive	L'aspect-MOP permet de modifier la logique de composition des aspects
SAFRAN [71]	Composants	Script Fscript	non-invasive	A la charge du développeur
Munnelly [18]	Objets	code	invasive	A la charge du développeur

La séparation des préoccupations transverses permet d'augmenter la réutilisabilité ainsi que la modularité des adaptations et permet de les composer. Il s'agit d'un premier moyen de gérer la variabilité de l'infrastructure et du système associé. Il n'est plus nécessaire de spécifier de manière ad-hoc toutes les configurations que nous souhaitons atteignables par le système. En effet, il est possible de composer dynamiquement des entités réutilisables pour des fonctionnalités clairement identifiées. Comme nous l'avons déjà évoqué, leur combinatoire est trop grande (voir non-bornée) pour que cela puisse être envisagé. Les deux grandes approches que nous avons présenté pour réaliser cette séparation des préoccupations ne sont pas incompatibles, au contraire elles peuvent être combinées et permettre ainsi la gestion des préoccupations transverses hétérogènes (FOP) comme des préoccupations transverses homogènes (AOP). Entre autre, l'AOP peut permettre une bonne mise en œuvre de la FOP. Dans le cadre de l'informatique ambiante, la gestion des préoccupations transverses hétérogènes peut autoriser l'adaptation et la prise en compte des dispositifs hétérogènes ou encore permettre la gestion de variantes d'adaptations pour une même fonctionnalité. D'un autre coté, la gestion des préoccupations transverses homogènes peut favoriser la gestion de la multiplicité des dispositifs composant l'infrastructure du système et permettre de dupliquer une adaptation en plusieurs endroits dans le système.

Comme nous l'avons vu, quelque soit l'approche utilisée, des interférences entre entités d'adaptations peuvent intervenir et les approches classiques pour gérer ces interférences reposent sur la définition explicite des dépendances entre entités d'adaptation, par exemple avec les notions de précédence ou de contrats. Ce type d'approche limite les capacités du mécanisme d'adaptation à prendre en compte l'imprévisibilité de l'environnement, de l'infrastructure et de l'ensemble des adaptations à mettre en œuvre. En raison de la grande combinatoire des adaptations à considérer, il peut vite devenir difficile de spécifier toutes ces dépendances comme d'en ajouter de nouvelles, limitant ainsi leur extensibilité. De plus, toutes ces dépendances ne peuvent être connues a priori. Quelque soit la technique utilisée pour réaliser cette séparation des préoccupations transverses, elle nécessite la mise en place d'un mécanisme, que nous appellerons de décision, définissant, parmi les adaptations déployées, quelles sont celles qui doivent être appliquées et de quelle manière les composer. Un tel mécanisme a nécessairement des répercussions sur les temps de réponse de l'adaptation.

2.5 Gérer la variabilité dans des environnements imprévisibles

Le mécanisme de décision a pour objectif de produire un plan de décision qui se compose de l'ensemble des adaptations à effectuer et qui peut être accompagné d'indications sur la manière de les composer. Cette décision se fait en fonction des informations qui lui sont données en entrée. D'une manière générale, dans le cadre de l'informatique ambiante, la décision fait la jonction entre la perception du contexte et l'adaptation. Le mécanisme de décision, puisqu'il permet de gérer la manière selon laquelle sont combinées les adaptations, peut limiter l'espace des reconfigurations atteignables par un système de reconfiguration. Il peut être plus ou moins flexible dans la manière dont il va composer ces adaptations. Dans la feuille de route du séminaire de Dagstuhl relatif au génie logiciel pour les systèmes auto-adaptatifs [158], il est exprimé que certaines approches définissent explicitement toutes les adaptations, tandis que d'autres dans une approche implicite les font émergées. Il est également possible de définir des approches dites hybrides explicites-implicites.

Les mécanismes les plus contraints devront spécifier, pour chaque situation, l'ensemble des adaptations qui doivent être réalisées et ce de manière exhaustive. Puisque tous les états du système sont connus, il est alors possible de construire les règles de décision de manière à ce que l'ensemble des adaptations associées à une situation soit consistant. Cette tâche peut être laissée au développeur ou réalisée de manière automatique. Le mécanisme d'adaptation est alors certain de produire une application cohérente. Ces approches sont le plus souvent conçues avec une stratégie « *top-down* » , c'est-à-dire que l'on partira de la définition d'un objectif abstrait que l'on raffinera jusqu'à spécifier exhaustivement les adaptations ainsi que les situations permettant de remplir cet objectif.

Dans les approches les plus flexibles, le mécanisme de décision laissera plutôt le système évoluer par lui-même dans l'optique, par la suite, d'influencer son développement pour qu'il atteigne certains objectifs. La gestion des conflits entre adaptations sera alors déléguée au mécanisme d'adaptation. En effet, le mécanisme de décision a pour objectif d'influencer ces adaptations de manière à faire converger le système vers des objectifs attendus. Cette stratégie de conception est également utilisée dans les approches travaillant sur des abstractions. Contrairement aux autres approches, celles-ci se baseront plutôt sur une stratégie « *bottom-up* » , pour laquelle, en fonction des opportunités offertes au système, on le construira et cherchera à l'influencer, on parle alors d'émergence. Toutefois, l'émergence, lorsque contrôlée, ne peut pas reposer exclusivement sur une stratégie « *bottom-up* » , mais plutôt sur un compromis entre les deux stratégies [159]. La définition et l'utilisation d'objectifs

autorise le contrôle du comportement du système. C'est la définition de cet objectif, qui est donc nécessaire au contrôle du système qui émerge, qui introduit le couplage entre les deux stratégies.

Entre ces deux extrêmes, se trouve un niveau intermédiaire dans lequel il est possible de travailler sur des types d'adaptations, c'est-à-dire d'offrir une abstraction permettant de généraliser ou de générer à la fois les processus de sélection des adaptations et indiquant de quelle manière les composer. Ceci permet de limiter le nombre de règles de décision à définir. Le mécanisme de décision pourra définir des types de contraintes selon lesquelles des types d'adaptation peuvent être réalisés. Par exemple, nous exprimerons le fait que le type d'adaptation permettant de gérer l'économie d'énergie et le type permettant de gérer les cas d'urgence ne sont pas compatibles. Comme dans les approches les plus contraintes, la conception de ces mécanismes repose sur une stratégie « *top-down* »

Plus le mécanisme de décision est contraint, plus l'espace des solutions fournies par le mécanisme d'adaptation est limité [159]. Inversement, moins il est contraint, plus il sera facile d'étendre le mécanisme de décision et d'ajouter des adaptations. Ces approches permettent donc plus ou moins de prendre en compte les trois axes d'imprévisibilité définis en section 1.2.2, et plus particulièrement l'imprévisibilité relative à l'ensemble des adaptations à mettre en œuvre. Il y a donc un continuum entre les approches les plus contraintes et les plus flexibles que nous allons étudier par la suite.

2.5.1 Approches explicites : spécifier toutes les configurations et adaptations faisant passer d'une configuration à une autre

Dans les approches explicites, le mécanisme de décision réduit au maximum l'espace de liberté laissé au système pour évoluer dans des états qui n'avaient pas été prévus à la conception.

Une première possibilité pour concevoir de tels mécanismes de décision réside dans la définition explicite des dépendances entre adaptations. C'est ce type d'approche que l'on retrouve classiquement dans la programmation par aspect. En effet, afin d'éviter les interférences entre aspects, des dépendances entre les entités sont exprimées. Nous avons vu que, traditionnellement, la notion de précédence permet de définir un ordre entre les aspects [16]. D'autres approches proposent d'utiliser des contrats entre les entités d'adaptation [12]. Il devient alors nécessaire, lors de chaque ajout d'une entité d'adaptation, de vérifier de quelle manière cette dernière peut se composer avec les autres. Le nombre de vérifications à réaliser peut atteindre jusqu'à 2^n, n étant le nombre d'entités d'adaptation. Ce nombre de vérifications n'est acceptable que pour des systèmes peu complexes, pour lesquels une fiabilité maximale doit être assurée. Il ne laisse alors que peu, voir aucune, place à l'imprévisibilité.

Une seconde possibilité consiste à incorporer dans le mécanisme de décision une énumération des adaptations qui doivent être appliquées dans une situation particulière. Il est possible, via des abstractions, de simplifier la spécification des dépendances entre entités d'adaptation même si la combinatoire peut être aussi grande. Nous allons présenter maintenant quelques travaux reposant sur une telle approche.

Bencomo *et al*, avec Genie [55], proposent une approche permettant la génération et les reconfigurations de systèmes adaptatifs à base de composants. Pour ce faire, deux types de modèles sont définis. Le premier permet de spécifier des reconfigurations tandis que le second décrit l'ensemble des transitions permettant de passer d'une configuration à une autre. Dans la FIGURE 2.13, le niveau 3 représente cet automate à état fini. Il est le moyen de modéliser la variabilité du contexte du système.

Chaque état représente une situation, et les transitions décrivent les changements dans l'environnement faisant passer d'une situation à une autre. Ces changements sont exprimés sous la forme de contraintes. A chaque état est associé une configuration (niveau 2 de la FIGURE 2.13) décrivant la variabilité structurelle du système. A partir de ces configurations sont générés des fichiers de configuration XML (niveau 1 de la FIGURE 2.13). Dans cette approche, toutes les situations, transitions et configurations sont définies explicitement. Le nombre de transitions à spécifier peut alors vite devenir énorme et par conséquent difficile à gérer et à faire évoluer.

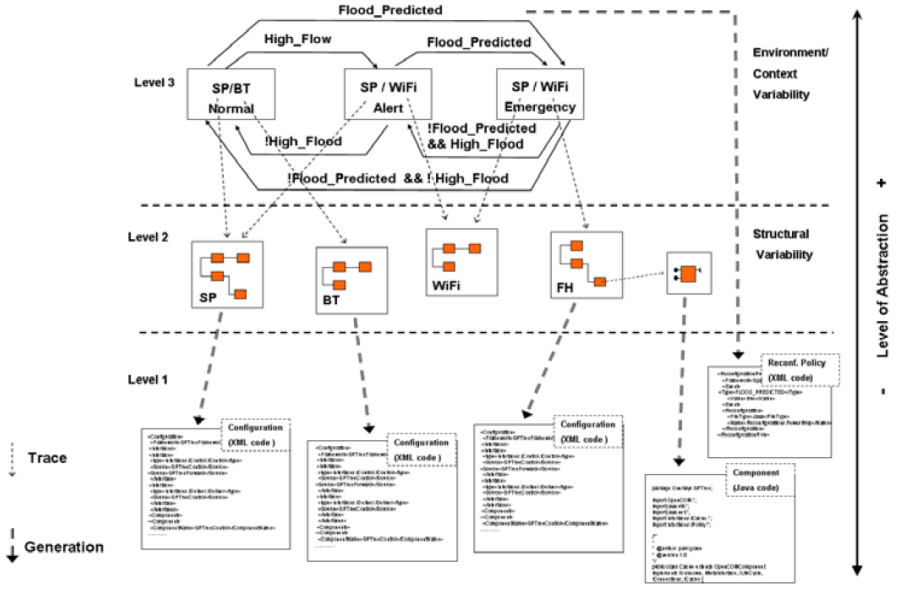


FIGURE 2.13 – Approche développée dans Genie [55].

Dans SAFRAN [71], la décision se fait dans un contrôleur se trouvant dans la membrane d'un composant Fractal. Nous avons vu que ce contrôleur lie les processus de perception des informations contextuelles et les processus d'adaptations à l'aide d'ensemble de règles ECA. Il faut donc spécifier explicitement l'ensemble de ces règles de décision et vérifier, lors de leur ajout, qu'elles sont compatibles entre elles. L'approche est difficilement extensible ; il peut d'ailleurs être envisagé de transformer un ensemble de règles ECA sous la forme d'un automate à état fini. Cependant, ces règles restent à un niveau local. Le composant, auquel elles sont attachées, pouvant être réutilisé de manière hiérarchique, les décisions de différents composants peuvent être combinées. Aucun mécanisme permettant de garantir la cohérence globale du système n'est présent.

Le projet RCSM (Reconfigurable Context Sensitive Middleware) définit et utilise des objets sensibles au contexte [116]. Des adaptations compositionnelles peuvent être réalisées sur ces objets. Ils consistent en deux parties : (1) une interface sensible au contexte et (2) une implémentation indépendante du contexte. Seule l'interface englobe la description de la sensibilité au contexte de l'application. Les conteneurs des objets sensibles au contexte communiquent ensemble pour obtenir les informations contextuelles qui modifieront la façon dont est implémentée l'interface de l'objet. Chacune de ces interfaces peut évoluer en fonction du contexte. Il est alors nécessaire de décrire explicitement, à l'intérieur de celle-ci, les conditions permettant l'activation ou non des méthodes définies. La variabilité et les capacités des objets sensibles au contexte sont donc explicitement

spécifiées. Dans les approches à composants, de telles interfaces pourraient également être définies.

Quelque soit le mécanisme choisi, cette approche a pour principal avantage de ne pas nécessiter de la part du mécanisme d'adaptation un système de gestion des conflits entre adaptations. La spécification explicite de toutes les règles de décision permet à la conception d'éviter ces conflits. De plus, comme cela est exprimé dans la feuille de route [158], plus la représentation du système est précise, plus il est possible de l'analyser. Cependant, le mécanisme de décision devient vite difficilement extensible et reste plus particulièrement adapté à des systèmes peu complexes, ayant des objectifs précis, pour lesquels la variabilité reste faible et l'imprévisibilité doit être réduite au maximum. D'autres approches permettent, à l'aide d'un ensemble restreint de règles de décision, de décrire autant de configurations et de transitions entre ces configurations. Moins il y a de règles, plus il sera facile de faire évoluer le mécanisme de décision. Si ces règles sont suffisamment abstraites elles peuvent être applicables dans plusieurs situations, parfois même à des situations qui n'avaient pas étés prévues. Cela permet de ne pas définir de manière ad-hoc les règles pour chaque situation. Les approches basées sur des types offrent de telles possibilités.

2.5.2 Augmenter l'abstraction pour limiter le nombre de configurations et de transitions entre ces configurations

Ces approches permettent, grâce à l'expression d'un ensemble réduit de contraintes, qui peuvent être abstraite, la génération d'un grand nombre de configurations du système. Celles-ci étant le résultat du produit croisé des différentes fonctionnalités du système qui vérifient les contraintes précédemment exprimées. Il s'agit finalement de générer un automate à états finis, à partir d'un ensemble réduit de règles. Grâce à l'abstraction offerte et au nombre limité de règles à analyser, il devient plus aisé que dans les approches totalement explicites d'ajouter de nouvelles contraintes et de prendre en considération la variabilité et l'imprévisibilité du système. Par exemple, une contrainte suffisamment générique peut s'appliquer dans plusieurs cas et parfois même dans certains qui n'avaient pas été prévus lors de sa conception (potentiellement à tort). Dans ces approches, il est également possible de définir des règles de bas niveau, par contre, plus les règles sont concrètes, plus on s'approche d'une définition explicite des contraintes.

Les travaux dans le domaine des lignes de produits dynamiques proposent de telles approches. Les lignes de produits sont une approche largement utilisée dans l'industrie et qui a rencontré de nombreux succès. Dans le domaine du logiciel, lignes de produits et développement orienté feature sont des notions fortement associées. Une ligne de produit se définit comme suit :

Définition 8 : Ligne de produits

« *A set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way* » [84]

Les lignes de produits permettent de développer des logiciels personnalisables comme dans des chaînes de production. Il s'agit d'un moyen d'introduire de la variabilité dans une chaîne de production logiciel [92]. Les différents éléments produits dans la chaîne peuvent être personnalisés selon les contraintes de production. Les lignes de produits sont donc motivées par une réduction des coûts de production grâce à une forte réutilisation d'éléments logiciels tout en minimisant leurs

temps de mise sur le marché. Les deux challenges dans les lignes de produits sont d'exploiter les points communs et de gérer la variabilité de ces éléments. Dans le domaine des lignes de produits, la variabilité se définie comme l'ensemble des hypothèses montrant comment les produits, membres de la ligne de produit, diffèrent [92]. Les **points de variations** sont les endroits où une personnalisation, une variation, peut être appliquée. Un **variant** est un élément logiciel qui peut être sélectionné pour implémenter un point de variation. Il s'agit d'une représentation d'un élément variable de la ligne de produit. Par exemple un pneu peut avoir plusieurs variants : un pneu pluie, un pneu neige ...

Les features models sont un moyen de décrire la variabilité des lignes de produits (FIGURE 2.14). Ils permettent à la fois d'exprimer des décompositions ainsi que des contraintes entre des ensembles de produits similaires [28].

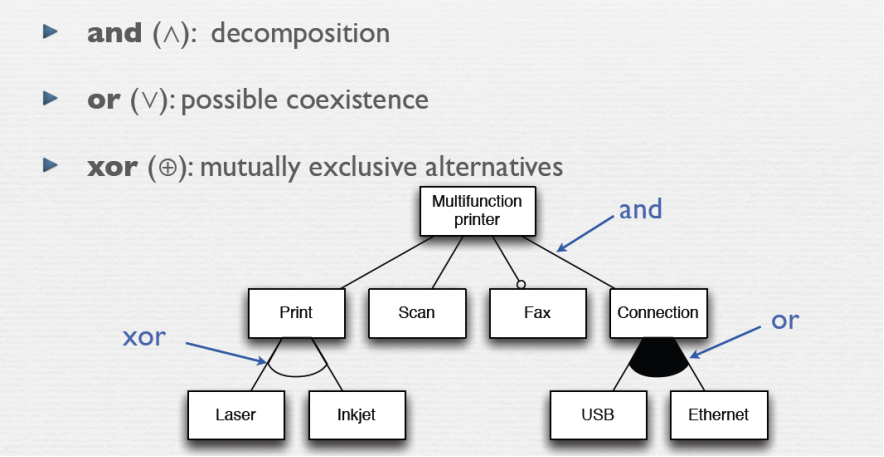


FIGURE 2.14 – Décompositions dans les diagrammes de features [28].

Les lignes de produit dynamiques proposent d'utiliser les techniques des lignes de produits à l'exécution. Les lignes de produits peuvent être adaptées à l'exécution en fonction des besoins utilisateur et de l'environnement. Il s'agit alors de sélectionner ou désélectionner des features à l'exécution, entraînant ainsi des reconfigurations logicielles en réaction à des changements intervenants dans le contexte de l'application.

Le framework CAPucine [99] (Context Aware Product Lines) utilise une telle approche pour réaliser des adaptations dynamiques d'application à base de composants. Une adaptation consiste alors à faire passer un produit d'une configuration à une autre. Un produit est modélisé comme un ensemble de features sélectionnés. Les features sont implémentés sous la forme d'aspects sur des modèles et la variabilité du système est modélisée sous la forme de diagramme de features. Le processus de décision dans cette approche repose sur trois grand mécanismes : (1) l'*adapter* qui a pour objectif de définir le nouvel ensemble de feature définissant la nouvelle configuration du produit, (2) le *validator* qui analyse si la configuration vérifie bien les contraintes exprimées dans le diagramme de feature et (3) le *script generator* qui génère, à partir des aspects sélectionnés, des scripts décrivant les modifications à porter sur la plateforme d'exécution sous-jacent (ajout/retrait de liaisons/composants). L'*adapter* prend en entrée la configuration courante du produit ainsi que des observables fournis par un *context manager*. Les aspects possèdent des observables indiquant quand ils doivent être tissés. Le processus de création d'une nouvelle configuration prend alors la

configuration courante et sélectionne (ajoute) les aspects qui peuvent être tissés et inversement pour ceux qui ne le peuvent pas. Ensuite, le validator examine si la configuration vérifie les contraintes définies dans le diagramme de feature et si les features peuvent être composées. Ce processus, qui peut être coûteux en temps, est préparé automatiquement avant le déploiement du système. A partir du diagramme de feature, une table exprime l'ensemble des configurations du produit. Dans cette table, est ensuite retiré l'ensemble des configurations ne respectant pas les contraintes du diagramme. Enfin, sont retirées les entrées de la table contenant des aspects qui interfèrent explicitement. Le validator au runtime vérifie si la configuration produite par l'adapter correspond à une entrée de la table.

Dans cette approche, il n'est donc pas nécessaire de définir explicitement toutes les configurations atteignables par le système. La combinatoire des règles à définir à la conception est donc nettement plus faible que dans les approches explicites. Par contre, la génération de la table de validation peut être une étape coûteuse en temps lorsque l'on souhaite étendre le mécanisme de décision en ajoutant ou modifiant le diagramme de feature. Le système peut évoluer uniquement dans les configurations définies dans la table de toutes les configurations autorisées.

B. Morin dans sa thèse [96] et dans le projet Diva propose également l'utilisation de diagramme de feature pour décrire la variabilité des système auto-adaptable. Comme précédemment, cela permet de réduire très fortement le nombre de règles de décision à définir à la conception. Dans cette approche divers types de modèles sont manipulés. Un méta-modèle de variabilité, décrivant les variants pouvant former une configuration du système est définie. Un méta-modèle du contexte permet de spécifier ce qui est pertinent dans le contexte du système. Un méta-modèle de règles de décision (ECA et orientée but) étend et réutilise les deux méta-modèles précédents. Enfin, un méta-modèle d'architecture permet de décrire des assemblages de composant. Ce méta-modèle peut être utilisé sur plusieurs plateformes à composant comme Fractal ou encore OSGi. Grâce à l'Aspect Oriented Modeling (AOM), les adaptations, et donc les features, prennent la forme d'aspects. Tous les raisonnements prennent en entrée et produisent des modèles de la variabilité, ces modèles permettent ensuite de sélectionner les aspects à tisser avant de reporter les adaptations sur l'application s'exécutant (FIGURE 2.15).

Comme dans l'approche précédente, un certain nombre de modèles doit être défini à la conception, mais la combinatoire des règles à décrire reste nettement plus faible que dans les approches explicites ce qui permet de concevoir avec cette approche des systèmes plus complexes. Cependant, comme précédemment, l'approche de spécification « *top-down* » ne permet au système que d'évoluer dans l'espace des solutions dérivé des spécifications de départ.

D'autres approches existent et ne sont pas basées sur les techniques du domaine des lignes de produits. Par exemple dans [121], un système de gestion des buts est proposé. Un but est réalisé par un ensemble d'actions. Un modèle de l'environnement appelé « domain model » est représenté sous la forme d'un automate à états finis. Cet automate décrit les états logiques dans lesquelles peuvent se trouver l'environnement et le système ainsi que les transitions entre ces états. Cette représentation du *domain model* est générée grâce au Labelled Transition System Analyser (LTSA) à partir d'un ensemble d'actions, de propositions et de contraintes. Les actions sont organisées hiérarchiquement et décrivent les actions que doit réaliser le système, par exemple : *loadBallToX*. Les contraintes expriment des pré-conditions qui doivent être remplies pour l'occurrence d'une action. Les propositions décrivent des actions terminales, c'est-à-dire qui permettent de remplir le but. La

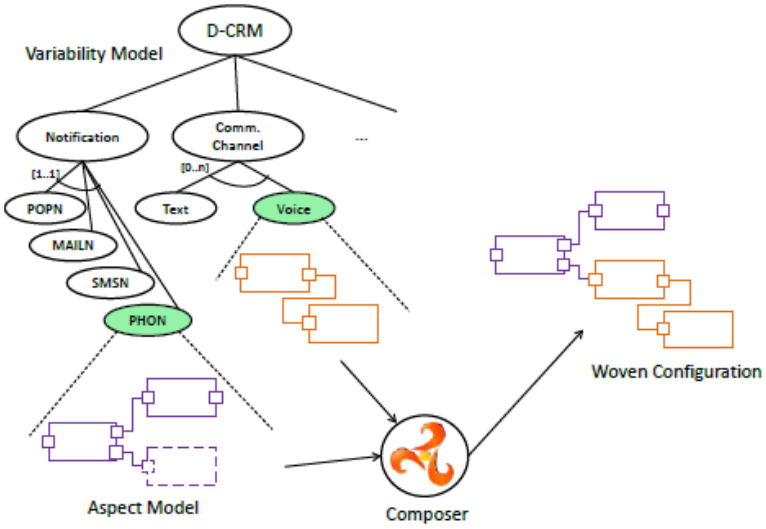


FIGURE 2.15 – Diagramme de feature pour sélectionner des aspects [96].

réalisation d'un but s'obtient par un enchaînement d'actions. Le système tente de trouver le meilleur chemin d'actions permettant d'atteindre l'objectif. Comme dans les approches inspirées par les lignes de produits, ce mécanisme permet la définition d'un nombre relativement restreint de règles abstraites dans l'optique de générer l'ensemble de l'espace des solutions d'adaptation.

Les approches explicites, comme celles basées sur des types, reposent sur une stratégie de conception *top-down*. Ces approches offrent un degré de liberté, en terme d'évolution du système, limité. Cependant, l'imprévisibilité du contexte peut nécessiter que le système s'adapte à des changements qui n'avaient pas été anticipés pour le mener dans des configurations qui ne l'étaient pas non plus. Dans [159], le constat suivant est réalisé : les stratégies classiques de conception *top-down* ont montré leurs limites. Parallèlement à cela, une conception totalement *bottom-up* ne semblent pas envisageable et mène trop souvent à des systèmes dont il est difficile de maîtriser le comportement. La solution se trouve donc dans un compromis entre les deux. Les approches *top down* sont trop contraintes tandis que les conceptions *bottom up* mènent souvent à des systèmes qui ne sont pas maîtrisables. Les approches se basant sur ce compromis, permettent de faire émerger de nouveaux système mais de manière contrôlée on parlera d'« *émergence contrôlée* » .

2.5.3 Vers de l'émergence contrôlée

Ces approches, tentent de limiter au minimum l'espace des configurations atteignables par le système. Elles proposent le processus de développement qui suit. L'infrastructure évolue dynamiquement, des dispositifs apparaissent et disparaissent modifiant les possibilités de configuration de l'application (par exemple : un téléphone est désormais disponible dans l'infrastructure du système ambiant). L'application, « par un heureux hasard » (*serendipitous*), évolue en fonction de cette infrastructure selon les opportunités offertes (par exemple : le téléphone est utilisé comme un bouton pour ajouter une nouvelle modalité d'interaction). A chaque opportunité une nouvelle configuration ou application est construite, émerge [150], au runtime avec un minimum d'interruption. Puisque

toutes les opportunités ne peuvent être envisagées, il n'y a alors pas de programmation explicite des applications, on ne recherche pas des dispositifs pour une fonctionnalité mais en fonction des dispositifs disponibles on construit ce que l'on peut pour l'objectif souhaité.

A la frontière entre approche explicite et par émergence, les projets MADAM [76] et MUSIC [109] proposent un compromis intéressant combinant les deux. Ils proposent un intergiciel de type « *planning-based* ». Il a la capacité d'adapter une application en fonction de changements intervenant dans son contexte opérationnel grâce à l'exploitation d'informations relatives à la composition ainsi qu'à des méta-données de qualité de service (QoS) associées aux composants de l'application [109]. Pour ce faire, ils se basent sur la notion de « *component framework* » qui sont des assemblages de types de composants, des assemblages de fonctionnalités. Ces patrons d'architectures, définis explicitement, réduisent fortement les capacités d'adaptation de l'approche. Par contre, ces assemblages peuvent, par la suite, être implémentés avec diverses instances de composants ou services dans une approche flexible. Le choix de l'instance de composant, permettant de réaliser une configuration, se fait à l'exécution par le mécanisme de *planning* dans l'optique de choisir la meilleure configuration dans une situation donnée. Chaque instance de composant pouvant correspondre à une fonctionnalité est réifié sur la forme d'un plan comprenant l'instance de composant ainsi que des méta-données associées à ce composant. Ensuite, des *properties predictors*, qui sont des fonctions, prédisent dans quelle mesure le composant remplit la fonctionnalité dans ce contexte. La fonction d'évaluation prend donc également des informations contextuelles comme entrées. Enfin, les différentes prédictions sont évaluées dans une fonction globale qui choisira la configuration la plus adaptée. Les différentes combinaisons sont ainsi toutes évaluées à l'exécution. Dans [108], l'approche est également appliquée aux aspects. Des méta-données peuvent également leur être associées. L'évaluation de l'impacte d'un aspect peut se faire soit dans la fonction globale pour savoir s'il doit être tissé, soit dans les prédictions.

L'émergence et l'auto-organisation doit pouvoir faire apparaître des comportements qui n'avaient pas été prévus à la conception. Les effets de l'émergence peuvent alors être positifs ou néfastes lorsqu'ils font apparaître des comportements non désirés ou par exemple instables. Le programme Allemand OC (**O**rganic **C**omputing) propose une architecture générique appelée « *observer/controler* » (FIGURE 2.16) [59] pour la conception de système organique permettant de réaliser de l'émergence contrôlée limitant au maximum ces effets néfastes. L'*observer* observe le système sous-jacent et transmet au *controler* une description de la situation contextuelle courante. Le *controler* évalue cette description et réalise un certain nombre d'actions afin d'influencer l'évolution du système dans l'optique de le faire converger vers un but particulier. C'est la définition de ce but qui rend cette conception à mi-chemin entre conception *top-down* et *bottom-up*. Les systèmes reposant sur cette architecture présentée en FIGURE 2.16, peuvent être totalement décentralisées, avoir une architecture hiérarchique ou centralisée. Dans le cas d'une architecture centralisée une seule boucle *observer/controler* est déployée pour l'ensemble des entités du système. Dans l'approche hiérarchique, des systèmes *observer/controler* peuvent être vus comme des entités et gérés par une boucle plus globale. Enfin, dans l'approche décentralisée, plusieurs systèmes observer/controler coexistent de manière autonome.

Comme cela est mis en avant dans [59], cette architecture offre de nombreuses similitudes avec la décomposition fonctionnelle classique des intergiciels sensibles au contexte et les architectures des systèmes autonomes [91] à savoir : perception, analyse, décision et réaction. Cependant, elle a pour principale différence d'être dirigée par des buts externes et de rapporter son statut à l'utilisateur. Celui-ci est alors en capacité d'intervenir à tout moment explicitement, par exemple en modifiants les buts [59]. Dans ces systèmes, la décision se fait donc dans le *controler*. Cette décision afin de

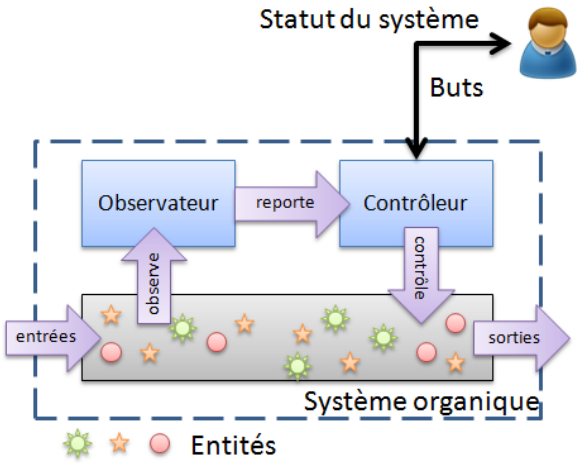


FIGURE 2.16 – Architecture « observer/controler » .

permettre l'émergence est la plus flexible et la plus générique possible. Pour ce faire, des mécanismes d'apprentissage ou encore des fonctions de fitness peuvent être utilisés. Souvent, à ces fonctions d'évaluation sont combinées des algorithmes évolutionnaires qui permettent leur génération ainsi que leur évolution.

Par exemple dans le projet européen Bionets [7], l'objectif est de manipuler des composants pour orchestrer des services [85] par assemblage de composants. Pour ce faire, il faut définir un comportement qui sera mis en place par des algorithmes évolutionnaires. Deux possibilités sont offertes : (1) créer une nouvelle composition et (2) améliorer une composition. La création d'une nouvelle composition se fait à partir d'une requête. Une requête définie les entrées, sorties, pré-conditions et post-conditions à respecter ainsi que l'utilisateur à l'origine de la requête. A partir d'une requête, un algorithme évolutionnaire décrit dans [85], permet de créer une composition. La FIGURE 2.17 présente un exemple de création d'une composition.

Améliorer une composition [78] consiste à remplacer des services qui la compose par de plus adaptés. Un service plus adapté peut être obtenu par composition de plusieurs autres services. Des fonctions de fitness permettent de définir comment un service remplit une fonctionnalité dans un contexte particulier. Les services en fonction des résultats peuvent alors être sélectionnés pour survivre, muter... Les services peuvent être combinés les uns avec les autres, au fur et à mesure que l'algorithme avance, pour former de nouveaux services agrandissant la population initiale. Pour évaluer cette fonction de fitness, des enregistrements sur des critères mesurables (mémoire, temps de réponses ...) sont créés. Ces enregistrements sont représentés sous la forme d'un modèle mathématique qui apporte le plus d'influence aux enregistrements qui arrivent le plus souvent. Par exemple dans la FIGURE 2.18 les enregistrement les plus fréquents quant à l'utilisation mémoire sont de 54 et 58 Mo. Cette modélisation a pour avantage, de ne pas nécessiter la conservation de tous les enregistrements et facilite les échanges d'enregistrements.

A partir de ces enregistrements, les services peuvent être comparés entre eux. Toutes les évaluations dans un même domaine se font avec une même fonction dont les bornes peuvent évoluer.

7. http ://www.bionets.eu

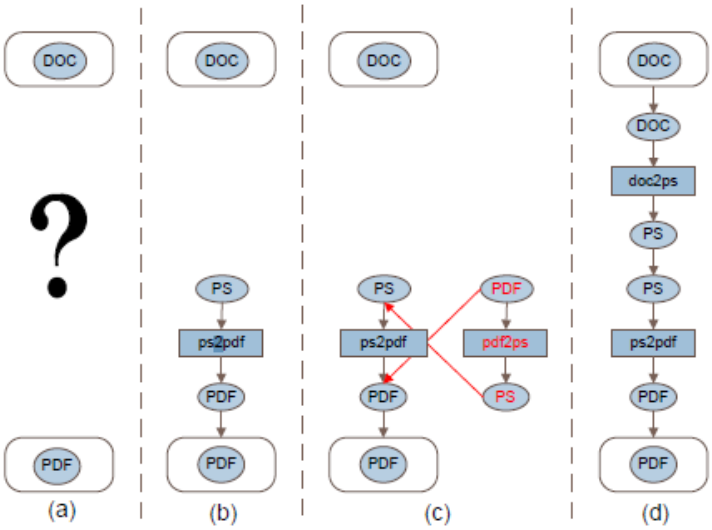


FIGURE 2.17 – Exemple de calcul de composition [85].

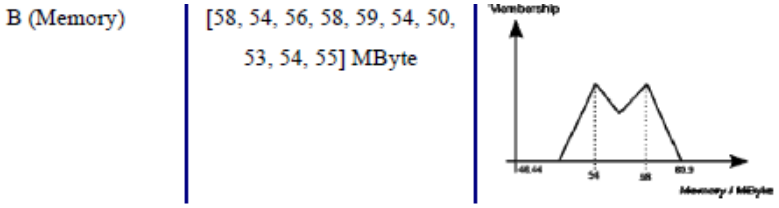


FIGURE 2.18 – Exemple d'enregistrement [78].

Un modèle basé sur de la logique floue est proposé pour représenter ces fonctions qui dépendent de préférences utilisateur. Les modèles d'enregistrement et d'évaluation sont composés de manière à pondérer chaque service en fonction de la fonction d'évaluation. Une fois évalués, les services sont triés en fonction de leur fitness. La composition est ensuite évaluée en réalisant une moyenne pondérée des services en fonction de leur influence.

Dans ces approches, l'espace des configurations atteignables par le système est moins limité et la spécifications des contraintes entre adaptations limitée, ce qui permet de faire émerger de nouveaux comportements. Une caractéristique importante de toutes ces approches est qu'elle permettent l'intervention de l'utilisateur, considérant que celui-ci est le mieux armé pour définir comment le système doit réagir ou évoluer lorsqu'il se trouve dans des situations qui n'avaient pas été anticipées à la conception. Cependant, il est important de noter que la durée d'une intervention utilisateur n'est pas prévisible et ne peut être définie comme bornée. Le principal inconvénient de ces approches est qu'elles n'offrent pas ou difficilement de garanties sur la bonne évolution du comportement du système. Si elles sont adaptées à des systèmes complexes et fortement variables, elles le sont moins pour des systèmes dont le bon comportement doit être garanti à tout instant.

Les différentes approches pouvant être utilisées dans des mécanismes de décision offrent différents compromis entre flexibilité et contrôle du système. Les approches explicites, sont les moins flexibles et par conséquent les moins apte à prendre en compte l'imprévisibilité du contexte mais aussi à évoluer. En effet, plus le système devient complexe et de grande taille, plus il devient difficile d'ajouter de nouvelles règles de décision tout en garantissant qu'il sera cohérent avec l'ensemble des autres règles. Par contre cette approche offre le contrôle le plus fin du système. Ce type d'approche est adapté aux systèmes dont les buts sont biens définis et précis qui nécessitent des garanties quant à leur bon fonctionnement. A l'extrême opposé se trouvent les approches pour l'émergence contrôlées. Ce sont les plus flexibles, elles permettent au système de faire émerger des configurations et parfois même des comportements qui n'avaient pas étés prévus tout en garantissant un contrôle minimal de la cohérence du système. Ces derniers sont les plus aptes à passer à l'échelle. Entre ces deux extrêmes, des approches permettent de trouver un compromis entre flexibilité, évolutivité et contrôle du système. Dans ces systèmes, bien souvent, l'ensemble des types de configurations est borné mais chacune d'entre elles peut être instancié de diverses manières.

Un mécanisme d'adaptation suffisamment générique doit pouvoir être utilisé dans les plan de réaction de tous les types de mécanismes de décision. Les adaptations doivent donc pouvoir être composées explicitement, dupliquées et par conséquent abstraites ou encore évoluer de manière opportuniste avec les variations de l'infrastructure pour pouvoir ensuite être influencées.

Enfin, il est important de noter que ces différentes approches pourraient être utilisées conjointement dans une même architecture en fonction de leurs caractéristiques. Ceci nous ramène à notre besoin de disposer d'un système capable de prendre en compte les multiples dynamiques d'évolution du contexte. Dans ce cas, plus qu'un mécanisme d'adaptation, c'est également une architecture permettant de combiner ces approches qu'il faut définir.

Dans la section suivante, nous ferons une synthèse des différentes approches, techniques, tirées de l'état de l'art, que nous considérons nécessaires à un mécanisme d'adaptation en informatique ambiante.

2.6 Synthèse et objectifs

Un mécanisme d'adaptation, pour être pertinent dans le domaine de l'informatique ambiante, devra tout d'abord disposer des caractéristiques que nous avons tirées de l'état de l'art qui lui permettront de respecter au mieux les contraintes présentées en introduction. Voici les caractéristiques, que doivet proposer le mécanisme d'adaptation, associées aux contraintes qu'elles contribuent à considérer.

Adaptation dynamique : L'adaptation doit être dynamique car il n'est pas envisageable d'arrêter l'application à chaque fois qu'une évolution est nécessaire. Le contexte peut évoluer fréquemment, s'y adapter peut être à l'origine de nombreuses adaptations du système. Une adaptation statique n'est donc pas envisageable.

↬ *contraintes adressées : sensibilité au contexte, temps de réponse.*

Adaptation compositionnelle : Le mécanisme d'adaptation doit permettre de réaliser a minima des adaptations compositionnelles. De cette manière, il sera possible d'intégrer à l'exécution de nouvelles briques logicielles. Ces briques peuvent être des entités se trouvant dans l'infrastructure logicielle de l'application. Ces entités ne sont pas nécessairement modifiables. D'autre part, puisqu'elles peuvent être mobiles, l'hypothèse de leur existence à un instant donné ne peut être ni émise ni même anticipée. Il reste envisageable de combiner adaptation paramétrée et compositionnelle. Toutefois, parce que l'on ne peut imaginer qu'il soit possible d'anticiper tous les cas d'adaptation, l'utilisation unique de l'adaptation paramétrée n'est pas envisageable.

↬ *contraintes adressées : imprévisibilité de l'infrastructure et des adaptations à mettre en œuvre, variabilité de l'infrastructure et des adaptations à mettre en œuvre, brique logicielles non modifiables.*

Traitements réalisés sur des représentations abstraites : L'adaptation doit, le moins possible, perturber le bon fonctionnement du système. Le résultat de l'adaptation doit être analysé avant son application. Lorsque les adaptations sont réalisées directement sur l'application s'exécutant via les mécanismes classiques de réflexivité, il n'est pas possible d'analyser le résultat des adaptations avant d'avoir modifié l'application. Or, ces modifications peuvent avoir des effets néfastes. L'utilisation de modèles de l'application à l'exécution permet de réaliser les traitements sur des représentations abstraites avant de les reporter dans l'application s'exécutant.

↬ *contraintes adressées : Continuité de service, respecter la dynamique de l'application.*

Séparation des préoccupations : L'adaptation doit être décrite et gérée comme une préoccupation transverse afin de réduire l'enchevêtrement du code, augmenter la réutilisabilité et l'indépendance des adaptations, gérer les préoccupations transverses homogènes et hétérogènes. Lorsque le code d'adaptation reste enchevêtré dans celui de l'application, il est difficile à faire évoluer et à maintenir. En particulier parce que ce code est souvent dupliqué en plusieurs endroits de l'application de manière transverse. D'autre part, les adaptations ne sont alors plus dans des entités indépendantes qui peuvent être réparties sur plusieurs acteurs.

↬ *contraintes adressées : variabilité et imprévisibilité, multiplicité des entités en jeu.*

L'ensemble de ces techniques et paradigmes peut être synthétisé sous la forme d'un peigne comme celui présenté FIGURE 2.19. Dans le cadre de l'adaptation paramétrée et compositionnelle, il apparaît d'après l'état de l'art que ces branches du peigne ne sont pas incompatibles mais peuvent être combinées.

Nous souhaitons ensuite que notre mécanisme d'adaptation offre des propriétés lui permettant d'être utilisé dans les approches de décision explicite comme dans les approches les plus flexibles dont la TABLE 2.2 propose une synthèse. Pour cela il doit proposer les propriétés suivantes :

- Nous devons pouvoir spécifier comment composer les entités d'adaptation. Le mécanisme d'adaptation doit garantir le **déterminisme** du résultat.
- Nous devons pouvoir ajouter des entités d'adaptation à l'exécution sans nous préoccuper des autres adaptations présentes, c'est-à-dire sans spécifier comment les composer avec les autres. L'ensemble des adaptations doit être **extensible** et les adaptations doivent être **indépendantes** les unes des autres. Le mécanisme d'adaptation doit inclure un système de gestion des conflits entre adaptations.

Toutes les contraintes logiques que nous venons d'identifier ont pour point commun de nécessiter un coût en temps qui peut être non-négligeable. L'adaptation compositionnelle, puisqu'elle met en jeu des mécanismes plus complexes que l'adaptation paramétrée, offre par conséquent des

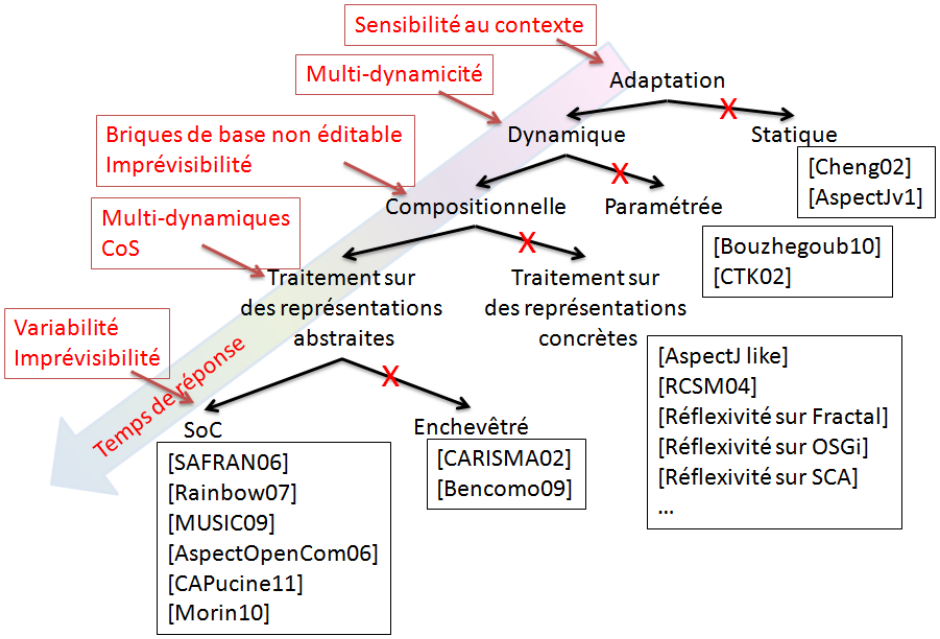


FIGURE 2.19 – Synthèse

temps de réponse plus important. De même, porter les adaptations dans un premier temps sur une représentation abstraite nécessite l'utilisation d'un mécanisme de synchronisation entre l'application s'exécutant et son modèle. Si cela peut permettre de réduire le nombre de modifications à réaliser dans l'application, le temps de calcul de ces adaptations peut être plus important. Enfin, la définition des adaptations dans des entités indépendantes et séparées de l'application de base nécessite de les composer à l'exécution. **A toutes ces propriétés nous ajoutons enfin la nécessité transverse de disposer d'un mécanisme d'adaptation offrant des temps de réponses suffisamment faibles et compatibles avec les dynamiques d'évolution de l'environnement y compris l'application et l'utilisateur.** Comme nous l'avons vu en introduction, la pertinence de l'adaptation ne doit pas être seulement logique à travers un ensemble de propriétés mais aussi temporelle. Des temps de réponse trop importants pourraient être à l'origine d'une désynchronisation entre le système et son environnement. Le système pourrait alors sombrer dans une forme d'indéterminisme qui consisterait à ne jamais être cohérent avec son environnement.

Enfin, nous avons vu que différents mécanismes de décision pourraient être utilisées conjointement dans une même architecture en fonction de leurs caractéristiques. Ceci correspond à notre besoin de disposer d'un système capable de prendre en compte les multiples dynamiques d'évolution du contexte. Dans ce cas, plus qu'un mécanisme d'adaptation c'est également une architecture permettant de combiner ces approches qu'il faut définir. L'architecture du système ambiant qui utilisera ce mécanisme d'adaptation doit également respecter un certain nombre de contraintes. Dans un premier temps, elle doit permettre d'utiliser une ou plusieurs approches de décision. Elle doit pouvoir, dans le cas des approches les plus flexibles, autoriser l'intervention de l'utilisateur ou de mécanismes dont les temps de réponses ne sont pas finis sans toutefois bloquer le système. Dans le cas ou plusieurs approches de décision sont utilisées, elle doit permettre de les organiser. Ceci doit permettre au système de respecter les diverses dynamiques de l'environnement et de l'application et par conséquent de garantir des temps de réponses minimaux lorsque nécessaire. Enfin, elle doit offrir

TABLE 2.2 – Synthèse des différentes approches existantes dans les mécanismes de décision.

Approche	**Nombre de règles, contraintes à définir**	**Garanties sur le bon fonctionnement**	**Prise en compte de l'imprévisibilité**	**Type d'approche de conception**
Explicite	Elevé, maximum	Elevées	Très difficile	Top-down
Abstraction et type	Variable, mais tend à être fortement réduit en comparaison avec les approches explicites	Variable, selon le niveau d'abstraction choisi. Plus faibles que dans les approches explicites	Possible mais limité, peut nécessiter d'intégrer de nouvelles règles parmi celles existantes.	Top-down
Emergence contrôlée	Variable, généralement le plus faible. Par contre, règles difficiles à définir	Plus on va vers de l'émergence, plus elles sont faibles	L'approche la plus apte	Compromis entre top-down et bottom-up

au système une continuité de service maximale.

Dans cette thèse, nous souhaitons proposer une approche qui permette la mise en place du continuum des approches de décision que nous venons d'étudier. Pour cela, nous proposons (1) une approche architecturale (2) ainsi qu'un mécanisme d'adaptation suffisamment générique pour gérer la variabilité et être utilisé de manière imprévisible. Nous allons maintenant étudier plus en détail les architectures classiques des intergiciels sensibles au contexte, nous présenterons dans la suite du document, notre approche architecturale. Nous verrons de quelle manière elle permet de répondre aux contraintes que nous venons de présenter. Nous préciserons dans cette architecture quel sera le rôle du moteur d'adaptation. A la suite de quoi nous étudierons en détail le mécanisme d'adaptation que nous proposons.

Deuxième partie

Contribution

CHAPITRE 3

Une architecture 4-couches multi-dynamiques

« L'architecture, c'est une tournure d'esprit et non un métier. »

Le Corbusier.

Sommaire

CETTE PARTIE présente en détail la contribution de la thèse. L'objectif de ce chapitre est de proposer une architecture permettant de combiner différents mécanismes de décision et de respecter les multiples dynamiques d'évolution de l'environnement. La manière selon laquelle ils peuvent être combinés permet de garantir la pertinence temporelle des adaptations (SECTION 1.2.4), en minimisant les temps d'interruption du système et en offrant des temps de réponse adaptés à la fréquence des variations des phénomènes observés. Pour ce faire, l'architecture s'inspire de travaux tirés de la robotique. Il est important de noter que cette thèse ne porte ni sur les mécanismes de décision ni sur la perception du contexte. Nous nous baserons sur des mécanismes déjà existants.

Dans ce chapitre, nous commencerons par étudier la décomposition fonctionnelle classique des mécanismes de prise en compte du contexte. Pour rappel, nous avons vu en SECTION 1.1 que la sensibilité au contexte est la capacité d'un système à prendre en compte les variations qui proviennent de son contexte et de s'y adapter de la manière la moins intrusive possible. Nous verrons ensuite les

architectures classiques des intergiciels sensibles au contexte. Après avoir identifié leurs limitations en terme de prise en compte des multiples dynamiques de l'environnement, nous verrons de quelle manière des travaux dans le domaine de la robotique, ensuite repris dans le domaine du self-management, y apportent des solutions. Nous proposerons alors une évolution de ces architectures pour notre cadre de l'informatique ambiante ainsi qu'une organisation des divers mécanismes de décision existants qui permet, entre autre, d'intégrer l'utilisateur dans le processus de décision.

3.1 L'architecture classique des intergiciels sensibles au contexte

Les mécanismes de prise en compte du contexte des intergiciels pour l'informatique ambiante reposent pour la plupart sur une décomposition fonctionnelle. Cette décomposition mène généralement à la mise en place d'architectures qui prennent la forme d'une séquence de ces fonctionnalités qui permettent difficilement de suivre les multiples dynamiques d'évolution de l'environnement.

3.1.1 Décomposition fonctionnelle classique des mécanismes de prise en compte du contexte

Les mécanismes classiques de prise en compte du contexte reposent sur une décomposition fonctionnelle motivée par la réutilisabilité et l'évolutivité. Elle repose généralement sur les grandes fonctionnalités que l'on retrouve dans [36] qui offre de très fortes similitudes avec le modèle MAPE-K de l'autonomic computing [91]. Dans un premier temps, cela consiste à recueillir des données brutes. Il s'agit de la phase de collecte des informations contextuelles. On parle également de cette étape comme de celle de la perception. Par la suite, ces données sont transformées en observables symboliques, en informations de plus haut niveau d'abstraction, par exemple à l'aide d'ontologies ou d'un moteur d'inférence. On déduit, par exemple, la proximité entre deux entités (savoir si elles sont dans un même bureau) à l'aide de leur localisation et d'un plan du bâtiment. A la suite de quoi, ces données servent lors d'une phase de décision, d'identification de la situation, qui produit un plan de réaction. Ce dernier est alors exécuté par le mécanisme d'adaptation aussi appelé mécanisme de contrôle. Bien entendu, ces étapes peuvent être affinées comme le montre la FIGURE 3.1

3.1.1.1 Observation du contexte

D'après Schilit [110], la sensibilité au contexte est la capacité d'un programme ou d'un dispositif à percevoir son propre état et les divers états de l'environnement. L'acquisition d'informations relatives au contexte s'effectue à l'aide de sondes, de capteurs. Ces capteurs sont les interfaces entre l'intergiciel et/ou l'application, et son environnement. Ils sont également parfois appelés *observateur de contexte*. Les données qu'ils transmettent sont appelés des *observables*.

Traditionnellement, lors de la réalisation de systèmes utilisant de tels mécanismes, les développeurs choisissent leurs sondes statiquement. La haute variabilité de l'infrastructure d'un espace ambiant rend cette solution de moins en moins acceptable puisqu'il devient impossible de définir à l'avance les capteurs qui seront à la disposition de l'application dès sa conception [40]. Les mécanismes de découverte de nouveaux capteurs sont une première solution permettant d'adapter les mécanismes de capture du contexte en fonction de l'environnement. Cependant, la découverte peut rester statique, puisqu'il peut s'agir pour un « *consumer* » de demander à un « *broker* » comment trouver un service désiré. Il s'agit donc d'un mécanisme de requêtes dans un annuaire connu. Par exemple, dans SOLAR [35] et Daidalos [43], les capteurs ou observateurs ne

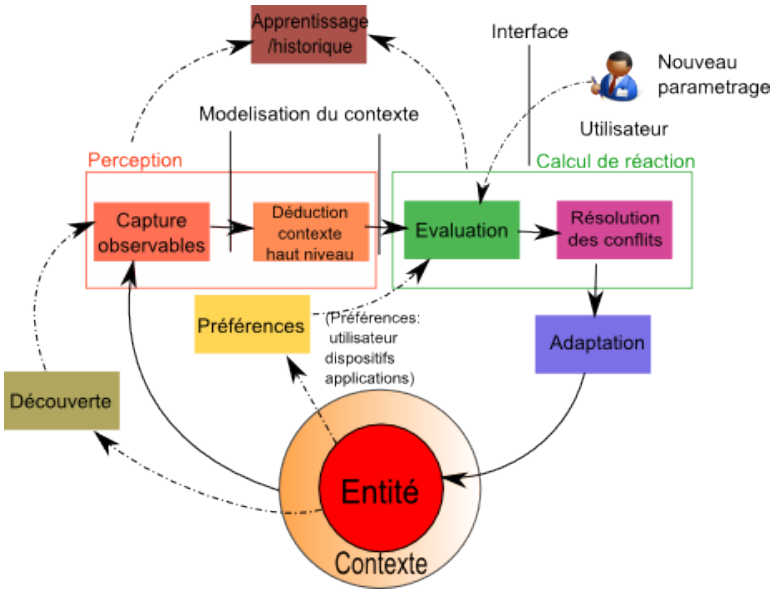


FIGURE 3.1 – Décomposition fonctionnelle de la prise en compte du contexte.

peuvent pas être découverts à partir d'une source inconnue. Ils doivent s'annoncer au gestionnaire de capteurs pour devenir une source d'information contextuelle dans le système. Ainsi, plus que la simple découverte, les notions d'apparition et de disparition de capteurs doivent être considérées. Elles mettent en jeu des mécanismes dynamiques qui ne peuvent pas être implémentés par des systèmes de requêtes mais à travers des annonces dynamiques (broadcasting). Donc, prendre en considération l'apparition et la disparition des capteurs nécessite de ne pas avoir recours à un annuaire.

Les communications avec les capteurs découverts peuvent alors s'opérer de deux manières. La première approche impose à la partie logicielle de prise en compte du contexte d'aller requérir l'information auprès des capteurs et ce, à intervalle régulier. Le système impose alors sa propre dynamique pour la collecte d'informations contextuelles. Dans la seconde, les informations peuvent être fournies à la plate-forme logicielle en mode *push*, par exemple, sous la forme de remontées de messages à chaque modification de l'état de l'observable. Cette approche reste la plus respectueuse de la dynamique de l'environnement puisqu'elle n'introduit comme latence, pour la prise en compte de cette information, que celle de l'envoi du message. Nous parlons alors de systèmes situés. La Context Toolkit [73] ou encore WildCat [69] reposent sur une approche asynchrone de notification.

3.1.1.2 Transformation des observables

Les données captées ne sont parfois pas toujours suffisantes pour alimenter les raisonnements des couches logicielles supérieures. Ainsi, l'étape de transformation en observables peut incorporer des mécanismes d'interprétation des informations captées dans l'optique d'en déduire des données de plus haut niveau ou encore d'améliorer la qualité des informations collectées. Il peut alors s'agir d'agréger (fusionner) ou encore de filtrer les informations remontées par les capteurs mais aussi de produire un raisonnement sur ces informations. Il peut être sémantique, par exemple, à l'aide d'ontologies, afin d'obtenir des informations plus pertinentes au bon niveau d'abstraction. Les données brutes ne sont pas nécessairement fiables, et l'objectif de ce processus peut être de détecter et de corriger des données erronées. Tout comme les données, les déductions produites dans cette étape du processus

de prise en compte du contexte ne sont pas nécessairement totalement fiables comme par exemple des déductions sur des critères sociaux ou encore des caractérisations comme identifier le ressenti d'un utilisateur. La notion de qualité de contexte (QoC) [33] peut alors être associée aux diverses informations contextuelles, ce qui permet de mesurer l'incertitude associée à une donnée. Le processus de transformation des observables n'est pas toujours mis en place dans les intergiciels sensibles au contexte. CARISMA [62], par exemple, ne dispose pas d'un tel mécanisme et ne peut collecter que des informations locales à la machine.

3.1.1.3 Modèles de représentation du contexte

Dans la plupart des intergiciels sensibles au contexte, c'est avant, ou parfois à la suite de cette transformation, qu'un modèle de représentation du contexte est en capacité d'être produit. Il est important de noter qu'il est possible d'utiliser plusieurs modèles du contexte. Une distinction fréquente est l'utilisation d'un modèle du contexte observé accompagné d'un modèle de contexte utilisé. Le premier est alors le résultat de l'observation du contexte tandis que le second est produit après interprétation. L'utilisation de modèles du contexte permet à plusieurs entités d'avoir un formalisme commun pour raisonner sur les mêmes bases [32].

De nombreux modèles pour représenter les informations contextuelles existent déjà dans la littérature et des états de l'art ont étés réalisés [42, 32] permettant d'évaluer leurs avantages et inconvénients. Les paires clé/valeur sont les représentations les plus basiques. Elles sont faciles à gérer mais n'expriment aucunes relations entre les informations contextuelles. Les n-uplets, moins basiques, représentent des informations hétérogènes et peuvent avoir une structure hiérarchique puisque des n-uplets peuvent contenir d'autres n-uplets. Ils ont pour avantage d'autoriser la distribution des données grâce aux *tuples spaces* [103]. Des modèles plus complexes existent comme les ontologies, les représentations graphiques comme UML ou des langages à balise comme XML. Ces modèles permettent plus facilement d'établir des relations entre informations contextuelles ou de leur associer des méta-données. Ils sont réutilisables et parfois composables. Enfin les modèles logiques ont pour principale avantage de combiner modélisation et analyse des informations. Ce type de représentation est utilisé lorsque l'on ne souhaite pas conserver de représentation de l'état du contexte. Aucune méta-donnée ou informations sur la validité ou la qualité des informations contextuelles ne peuvent leur être associées. Le modèle est alors léger et facile à échanger. Par exemple, dans le projet Bionets [102], des fonctions de logiques floues sont utilisées. Parfois, plusieurs types de modèles sont combinés. C'est le cas dans le projet Continuum, une base de connaissance qui repose sur une ontologie [31] est utilisée en complément d'un modèle logique [105].

3.1.1.4 Analyse du contexte et production d'un plan de réaction

Une fois collectées, les données relatives au contexte doivent être évaluées afin de calculer de quelle manière réagir. Le processus de décision, après avoir analysé le contexte courant, par exemple en se basant sur une représentation de ce dernier, a pour objectif de produire un plan de réaction. Il est aussi appelé processus d'identification d'une situation. Il a pour objectif d'identifier la ou les situations pertinentes dans lesquelles se trouve l'application et de proposer une adaptation en conséquence. Généralement, il spécifie « quand », « quelles » et « comment » les adaptations doivent être appliquées mais aussi parfois « où ». Ce mécanisme permet également d'identifier les changements faisant passer d'une situation à une autre. Nous avons vu en SECTION 2.5 divers mécanismes de décision.

L'obtention d'informations contextuelles peut être réalisée selon deux stratégies : *push* et *pull*. Avec la stratégie *push*, le processus est en attente d'événements l'informant de changements de contexte. En fonction de ces événements, le système réagit et propose une adaptation. Dans une stratégie *pull*, le mécanisme requière régulièrement des informations contextuelles. Par exemple, cela peut se faire en consultant périodiquement, un modèle du contexte dans un gestionnaire ou un serveur de contexte. En fonction des informations obtenues, il pourra déclencher des adaptations. Cette stratégie est par exemple utilisée lorsque le mécanisme de décision souhaite faire des requêtes à une base de connaissance. Dans Daidalos [43], un *sensor manager* met à jour les informations contextuelles de ces capteurs dans une base de données du système. Les changements qui sont apportés ne seront détectés que lors de la prochaine requête à la base. Cela nécessite la mise en place d'un système de fenêtre pendant lequel le système n'est pas sensible aux changements intervenant dans son contexte. Ceci implique que certaines modifications ne sont pas détectées ou le sont trop tardivement. La latence introduite, vient augmenter l'incertitude naturelle des observables. Bien entendu, les deux approches peuvent être combinées. Par exemple, un mécanisme peut être notifié d'un changement dans le contexte, afin qu'il soit capable d'évaluer si ce changement nécessite la mise en place d'une adaptation, il peut aller requérir auprès d'un gestionnaire de contexte un complément d'information.

La complexité de ces mécanismes est fortement dépendante des capacités du système à collecter et à interpréter les informations contextuelles. Mais les mécanismes de décision peuvent aussi disposer d'interfaces homme-machine pour leur déclenchement et leur configuration par l'utilisateur. D'autre part, si les observables courants sont à la base de cette évaluation, la possession d'un historique des contextes observés et des actions effectuées peut entrer en compte dans le choix d'une adaptation pour une situation donnée. Un historique permet entre autre de raisonner sur le contexte courant dans le cas où les informations viendraient à manquer. De plus, l'étude de l'observable doit être couplée à celle des différentes préférences fournies par l'utilisateur et qui sont un complément d'informations. Enfin, l'utilisation de techniques d'apprentissage peut être également envisagée. De tels mécanismes permettraient, entre autre, de réaliser des adaptations pro-actives. Le système serait alors capable de déduire à l'avance les adaptations qui doivent être réalisées. Bien qu'il soit largement admis que l'utilisation d'historique ou de techniques d'apprentissage offrent de nombreux avantages [39], ils ne sont que peu utilisés [34]. La phase d'évaluation peut donc être vue comme la détection, la reconnaissance d'une situation déclenchant une adaptation de l'application.

Nous retrouvons cette décomposition fonctionnelle dans la plupart des intergiciels sensibles au contexte. Bien souvent, celle-ci est directement transposée dans leur architecture. Nous allons maintenant étudier les architectures classiques des intergiciels sensibles au contexte. Nous observerons qu'elles peuvent difficilement suivre plusieurs dynamiques et donc répondre pleinement aux contraintes de l'informatique ambiante vues dans l'introduction.

3.1.2 Des architectures verticales

La décomposition fonctionnelle que nous venons de voir est souvent à l'origine de la mise en place d'architectures dites « verticales ». Une telle architecture se compose d'un ensemble de couches superposant diverses fonctionnalités. L'approche classique de conception de ces architectures consiste d'abord à identifier les fonctionnalités souhaitées puis leur enchaînement. Ensuite, les données traitées par ces fonctionnalités sont spécifiées ainsi que les entrées/sorties de chacune d'elles. C'est seulement

une fois toutes ces spécifications définies que le système est réalisé.

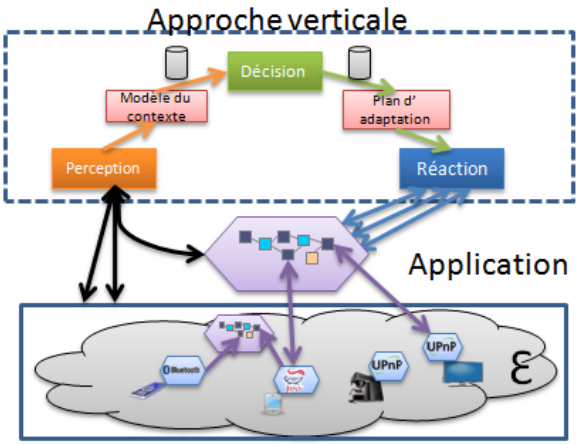


FIGURE 3.2 – Architecture verticale.

Cette approche de spécification logicielle entraîne naturellement une implémentation des diverses fonctionnalités de la décomposition sous la forme de grands blocs monolithiques et implique communément la mise en place d'architectures centralisées [115]. Lorsque l'on souhaite mettre en place des systèmes complexes et de grande taille, cette centralisation introduit des goulots d'étranglement. Dans le cadre de la prise en compte du contexte, les processus les plus touchés par cette centralisation sont ceux de décision, de représentation et de transformation du contexte. Quelque soit le changement de contexte considéré, tous les traitements sont alors dépendants de la dynamique de ces mécanismes. Par conséquent, les systèmes deviennent mono-dynamique, c'est-à-dire qu'ils réalisent les traitements comme des séquences d'opérations de capture, décision puis action (FIGURE 3.5). Cela peut avoir pour conséquence une désynchronisation entre l'environnement et le résultat de l'adaptation, faisant ainsi sombrer le système dans une forme d'indéterminisme (SECTION 1.2.4).

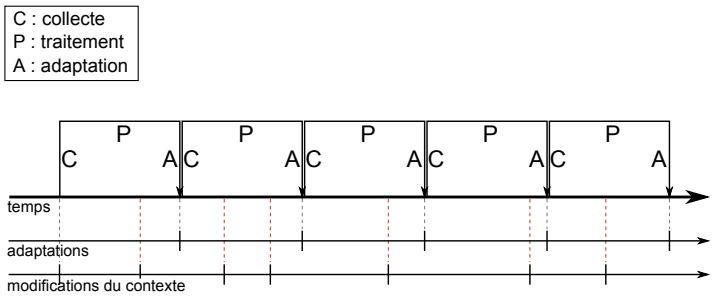


FIGURE 3.3 – Rythme des adaptations dans une architecture verticale.

Toutefois, ces architectures verticales peuvent avoir diverses granularités en fonction des entités auxquelles elles sont attachées, qu'il s'agisse d'entités locales comme un composant ou du système global. Par exemple, Gaia [106], un intergiciel sensible au contexte, permet la gestion d'espaces ambiants. Il prend la forme d'un système d'exploitation dans lequel tous les observables sont stockés dans une même entité : le « *context file system* ». Il en va de même pour SAFRAN [71] à une échelle plus locale. En effet, à un composant, peut être associé un contrôleur, contenant l'ensemble des règles

de décision, qui fait la liaison entre WildCAT et les adaptations. Ceci s'applique également aux intergiciels Rainbow [80], Genie [55] ou encore aux approches présentées dans [98] ou [63]. Enfin, certains travaux, plus particulièrement orientés vers la collecte d'informations contextuelles, c'est-à-dire ne proposant pas de mécanisme d'adaptation, offrent des mécanismes totalement décentralisés permettant de poursuivre plusieurs dynamiques comme CoWSAMI [53] ou encore SOCAM [83].

Les architectures verticales permettent donc difficilement de prendre en compte les multiples dynamiques d'évolution de leur contexte. Il devient alors difficile de réagir avec des temps de réponse adaptés à chaque phénomène observé. En particulier, il n'est pas aisé de réagir au plus vite aux changements nécessitant des temps de réponse faibles. Ces problématiques s'étaient également posées dans le domaine de la robotique. Les architectures réactives, inspirées par le comportement animal, puis les architectures sur 3 niveaux y ont apportées des solutions.

3.2 Une architecture tirée de la robotique : l'architecture 3T

Dans le cadre de la robotique, les architectures réactives, aussi appelées architectures horizontales, permettent la conception de systèmes capables de prendre en compte les multiples dynamiques d'évolution de l'environnement. A l'origine, le constat était le suivant : lorsque les systèmes sont découpés comme une séquence de processus utilisant une représentation interne de leur environnement, il leur est difficile de suivre plusieurs dynamiques. Les traitements, ainsi que la maintenance de cette représentation, peuvent mener à une désynchronisation entre l'environnement et sa représentation [120], de telle manière que la réponse apportée n'est plus en adéquation avec l'environnement. Les architectures horizontales apparues dans le milieu des années 80, avec l'architecture de subsumption de R. Brooks [118], reposent sur une décomposition comportementale qui apporte une solution à cette limitation. Malheureusement, nous allons voir qu'elle souffre également de limitations que d'autres architectures de la robotique, appelées « *3-Layers* » ou parfois « *3T* », se proposent de solutionner. Nous allons maintenant présenter cette décomposition comportementale puis les architectures 3-layers de la robotique. Ensuite, nous étudierons comment ces dernières ont pu être portées dans le domaine du self-management. Enfin, nous présenterons notre évolution de ces architectures pour le cadre de l'informatique ambiante.

3.2.1 A l'origine : une décomposition comportementale

Les architectures horizontales sont tirées du monde de la robotique et de l'intelligence artificielle. Il s'agit de spécialiser un cœur, minimal, d'application avec des composantes particulières (horizontales) [117]. Les composantes sont indépendantes les unes des autres et s'exécutent en parallèle. Chaque composante peut alors être connectée au monde via des capteurs et agir sur son environnement via des actionneurs. Ce type d'architecture introduit une nouvelle approche de décomposition : « la décomposition comportementale ».

D'après Bryson [119], la décomposition comportementale est une approche architecturale qui décompose l'intelligence en termes de comportements tels que manger ou marcher, plutôt qu'en des processus génériques comme planifier ou observer. Un comportement correspond alors à une composante horizontale et des activités peuvent se composer d'ensembles de comportements managés, faisant ainsi émerger un nouveau comportement global du robot. Une contribution forte de ces approches est de ne pas voir le système comme une séquence de processus mais plutôt comme

une parallélisation de processus et donc de comportements qui, ensemble, peuvent produire une activité cohérente. Il s'agit en réalité de décomposer le système en comportements (composantes horizontales) et ainsi, de décomposer des comportements complexes en de plus simples avec une stratégie « divide and conquer ». Les comportements peuvent avoir différents niveaux de complexité et ne pas être connectés au monde via des mécanismes de perception. Ainsi, le comportement le plus simple sera sans perception ni état [119] (par exemple un comportement pour un robot qui serait « rouler en continu »).

Ces architectures telles que définies par Brooks, respectent les caractéristiques suivantes :

- Chaque composante capture directement dans l'environnement ce qui est pertinent pour elle de manière à ce qu'elles puissent mettre en œuvre leur comportement.
- Il n'y a pas de représentation globale de l'environnement pour une approche décentralisée : « le monde est lui-même sa meilleure représentation » [118]. Les seules données exactes sur l'environnement sont les données obtenues immédiatement par les capteurs. Moins il y a d'états internes (de représentations de l'environnement), plus le comportement pourra être synchronisé avec l'environnement [120].
- Ces architectures comprennent de nombreuses composantes horizontales de faible complexité pour une réactivité maximale.

Les architectures horizontales requièrent un mécanisme de coordination (un gestionnaire) afin de combiner les données générées par chaque composante pour obtenir un comportement final global cohérent. Dans les premières architectures horizontales de Brooks [118], les « subsumption architectures », il s'agit d'un mécanisme de subsomption [1], c'est-à-dire qu'un comportement de niveau N+1 peut subsommer le rôle des comportements inférieurs en supprimant leurs sorties. Ce mécanisme est intéressant car peu complexe et peu coûteux en énergie ainsi qu'en mémoire [123]. A l'origine, dans cette architecture, pour garantir la bonne indépendance des niveaux et leur parallélisation, les différents niveaux étaient conçus comme des circuits différents. Plus le mécanisme est simple, plus sa réactivité est élevée et plus il pourra être mise en œuvre souvent. Il aura une influence d'autant plus grande sur le système. De plus, des temps de réponse faibles permettent d'avoir des comportements les plus synchronisés possibles avec leur environnement. A partir de ces caractéristiques, nous pouvons définir une activité comme un ensemble managé de comportements (FIGURE 3.4).

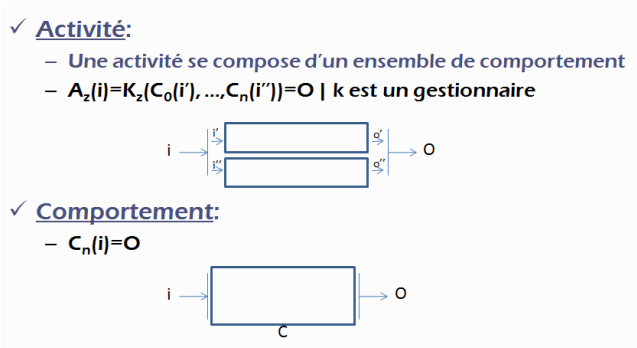


FIGURE 3.4 – Définition d'une activité dans une approche comportementale.

L'organisation des comportements n'est pas nécessairement basée sur le principe de subsomption. Différentes hiérarchies de comportements existent.

1. Raisonnement par lequel on met une idée sous une idée plus générale

3.2.1.1 Hiérarchies entre comportements

Les comportements peuvent être organisés selon une hiérarchie dictée par leurs temps de réponse, en quelque sorte une hiérarchie concurrente (FIGURE 3.5). Il n'y a pas de priorité entre les comportements et donc pas de mécanisme de priorité dans le mécanisme de coordination. L'importance entre les comportements se fait alors en fonction de la fréquence à laquelle ils peuvent s'appliquer. Plus cette fréquence est élevée, et donc plus son temps de réponse est faible, plus un comportement peut être appliqué. Cette approche a pour avantage d'offrir une très forte indépendance entre les comportements. Par contre, plus le nombre de comportements déployés dans les systèmes est important, plus la cohérence globale de l'activité du système est difficile à gérer. Dans le cadre de la robotique, cela est envisageable car l'ensemble des comportements est fixe et connu à l'avance.

Les comportements peuvent également être organisés selon une hiérarchie dictée par leurs temps de réponse couplée à un mécanisme de priorité (FIGURE 3.6), c'est-à-dire qu'un comportement peut être prioritaire sur un autre. Les comportements sont cependant toujours vus comme des boites noires. Il y a alors plusieurs manières de mettre en œuvre ce type de hiérarchie. Le mécanisme de contrôle peut être vu comme un ordonnanceur ou encore comme un système de contrôle des communications comme cela est le cas dans l'architecture de subsomption. Les traitements des comportements sont donc toujours indépendants les uns des autres mais il existe désormais des dépendances entre niveaux de comportements. Comparée à l'approche précédente, l'utilisation d'un mécanisme de priorité offre un médium, qui peut être peu coûteux en temps, pour gérer la cohérence de l'activité du système. Par contre, ajouter un comportement devient plus complexe et il est nécessaire de connaître tous les autres comportements afin de savoir quelle est la priorité du comportement décrit. On retrouve également cela dans les adaptations logicielles respectant le principe de la séparation des préoccupations, avec des notions de priorités ou de contraintes entre entités.

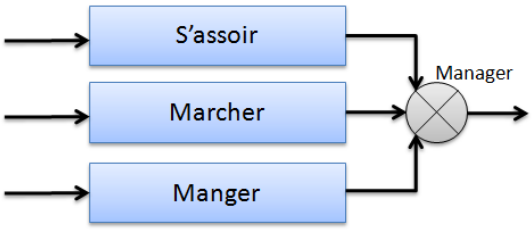


FIGURE 3.5 – Hiérarchie dictée par les temps de réponse des comportements.

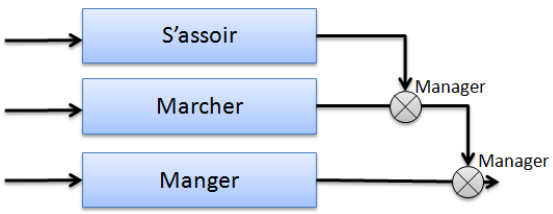


FIGURE 3.6 – Hiérarchie dictée par les temps de réponse des comportements avec priorité.

L'idée de ces architectures est d'être extrêmement réactive aux changements de l'environnement et de permettre aux robots d'évoluer dans des environnements fortement dynamiques, avec plusieurs dynamiques. Comparées aux architectures verticales, elles permettent de mieux maîtriser les dynamiques de l'environnement mais aussi de l'application. En effet, le traitement réalisé par une composante horizontale n'est effectué que lorsque cela est nécessaire. De plus, celui-ci est le plus rapide possible puisque chaque composante ne traite que ce qui est pertinent pour elle. Les diverses composantes horizontales étant indépendantes les unes des autres, elles offrent leur propre dynamique qui n'est plus tributaire des autres traitements (FIGURE 3.7). Il devient alors possible d'écrire des composantes avec des niveaux de complexité variés afin de respecter, au mieux, les dynamiques imposées par l'environnement et l'application.

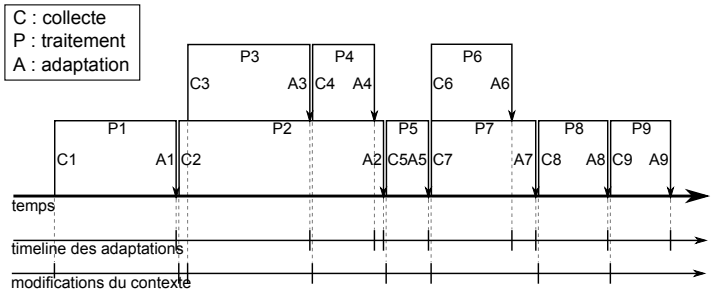


FIGURE 3.7 – Rythme des adaptations dans une architecture horizontale.

Cependant, la conception de tels systèmes n'est pas aisée et l'indépendance des comportements entre eux ne facilite pas la réutilisation du code qui les compose. Ces architectures restent difficiles à concevoir et peuvent être trop limitantes, et ce, en particulier lorsque l'on ne met en place que des comportements ne disposant d'aucune représentation interne de l'environnement. En robotique, ces architectures sont alors reprises et évoluent afin d'être organisées dans une architecture à trois niveaux. Cette architecture sera ensuite reprise dans le domaine du logicielle et appliquée au « *self-management* » puisque les comportements des robots présentent une forte analogie avec les boucles de contrôle des systèmes autonomes. En effet, les comportements des robots, aussi appelés « reactive plans », mettent en jeu une chaine de traitement reposant sur trois grands mécanismes : sense-plan-act, qui sont similaires aux fonctionnalités que nous retrouvons dans les boucles de contrôle des systèmes autonomes : monitor, analyse, plan, execute.

3.2.2 D'une architecture pour la robotique ...

Dans [120], Gat identifie des limitations aux architectures reposant sur une décomposition comportementale [2]. Il met en avant les points suivants :

1. Les architectures semblables à celles de Brooks requièrent une forte connaissance des différents niveaux qui la compose lors du développement d'un niveau. Modifier l'un d'entre eux impacte bien souvent plusieurs niveaux. Le comportement global résultant est souvent difficile à prédire. La structure figée des relations entre comportements n'est pas assez flexible.
2. Dans ces architectures, les comportements doivent être rapides et ne peuvent se baser sur un modèle ou un état interne représentant le monde. Ceci implique des restrictions sur les possibilités de décision d'un comportement.

En se basant sur le fait que la plupart des architectures des robots dans la littérature reposent sur trois composantes, il propose un modèle architectural sur trois niveaux appelée architecture 3T [120] et répond ainsi aux limitations des architectures horizontales. Les architectures trois niveaux se proposent d'organiser les algorithmes en fonction de leur gestion des états du monde et donc de la complexité de leur modèle de l'environnement. Les trois niveaux organisés comme en Figure 3.8 proposent les trois types de gestion des états de l'environnement suivants :

1. Sans état : *skill layer* (contrôleur) ;
2. État passé : *sequencing layer* (séquenceur) ;

2. Dans ce papier, le terme de *reactive plan* est utilisé pour parler d'un comportement en référence à l'approche Sense-Plan-Act (SPA)

3. État futur, déductions : *planning layer* (délibérateur).

Le premier niveau, sans état, est un mécanisme de rétrocontrôle réactif, le second propose un mécanisme d'exécution de *reactive plan* (comportements) tandis que le troisième est dédié aux calculs coûteux en temps. Les niveaux sont parallélisés et se trouvent dans des processus différents.

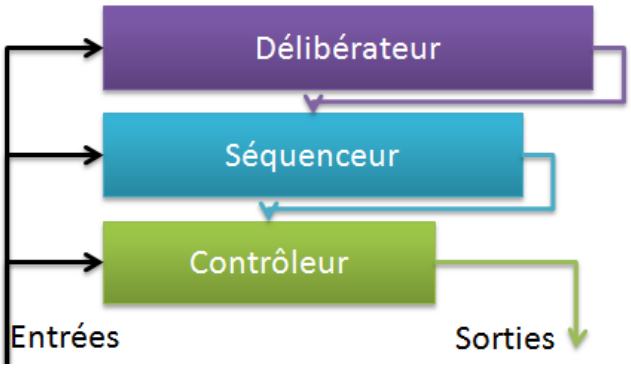


FIGURE 3.8 – Architecture 3T.

Le contrôleur consiste en un ensemble de threads implémentant des boucles de rétrocontrôle très fortement couplées aux capteurs et actionneurs du robot. Ces boucles de rétrocontrôle sont des comportements très simples qu'il est possible de composer afin de produire une activité plus complexe. Ce sont les comportements simples similaires à ceux que nous pouvions retrouver dans l'architecture de Brooks comme avancer et éviter les obstacles. Ces comportements peuvent être changés au runtime. Ils doivent être très réactifs et sans état. La manière dont ces comportements sont gérés les uns par rapport aux autres se fait via le séquenceur.

Le séquenceur a donc pour objectif de sélectionner les comportements qui doivent être utilisés par le contrôleur. Il peut également paramétrer ces comportements. Pour ce faire, plusieurs approches existent, la plus classique consiste à réaliser un automate à états finis définissant toutes les activités que peut réaliser le robot. Le séquenceur doit offrir des temps de réponse faibles afin de permettre l'ajustement du comportement d'un robot à une nouvelle situation de manière suffisamment rapide. Les traitements réalisés doivent donc être simples et offrir des temps de réponse adaptés.

Le délibérateur s'occupe des traitements complexes dont les temps de réponse sont importants. Le délibérateur peut produire des plans pour le séquenceur ou encore répondre à ces questions. Il est par exemple utilisé pour le traitement d'informations complexes comme des analyses d'images.

3.2.3 ... à une architecture pour le self-management

Kramer *et al* [122, 112, 121] proposent de s'inspirer du modèle architectural proposé par Gat pour en proposer une adaptation dans le domaine logiciel du *self-management*. Le constat de départ est le suivant : la robotique a déjà largement contribué dans le domaine des systèmes autonomes et les boucles de contrôle de la forme perception, décision, réaction des systèmes autonomes sont semblables au sense-plan-act de la robotique. Il est important de noter que dans ce modèle architectural, les niveaux ne sont pas nécessairement centralisés.

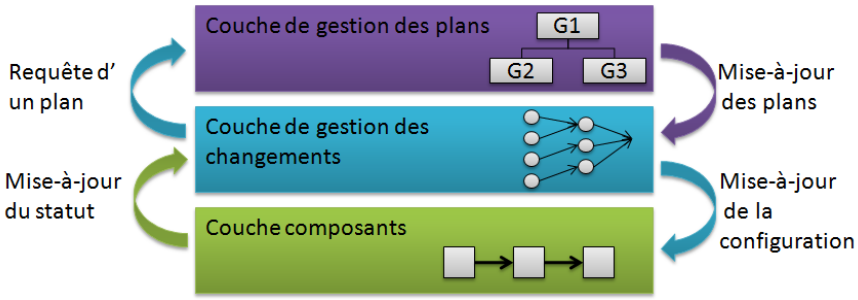


FIGURE 3.9 – Portage de l'architecture 3T pour le self-management [112].

Le niveau « *component control* » est le premier niveau de l'architecture ; il correspond, dans l'architecture 3T, au *skill layer (contrôleur)*. Si, en robotique, il s'agissait de mettre en place des boucles de rétrocontrôle entre les capteurs et actionneurs du robot, sa transposition ici consiste en des assemblages de composants. La plateforme d'exécution de ces composants doit être adaptative, c'est-à-dire qu'elle doit offrir des capacités d'introspection pour le monitoring et doit également permettre l'ajout/retrait de composants ou liaisons. Les comportements des composants peuvent également être modifiés à l'aide de paramètres. Il est identifié comme caractéristique importante, la capacité de ce niveau à identifier qu'un assemblage n'est pas adapté à la situation courante et qu'il doit être capable de faire remonter cette information au niveau supérieur [122].

Le niveau « *change management* » est le second niveau de l'architecture. Il est le pendant du *sequencing layer (séquenceur)* de l'architecture de Gat. Ce niveau a pour objectif d'adapter l'assemblage de composants soit en réponse aux changements détectés par le niveau inférieur, soit en raison du déploiement d'un nouvel objectif par le niveau supérieur [122]. Ce niveau se compose d'un ensemble de plans d'adaptations qui peuvent supprimer, ajouter, échanger des composants ou liaisons ou encore modifier des paramètres de composants. Ces plans sont déclenchés en réponse aux notifications du niveau inférieur et sont déployés par le niveau supérieur.

Le niveau « *goal management* » est le troisième étage de l'architecture qui peut être comparé au *planning layer (délibérateur)* du modèle 3T. Il produit des plans d'adaptation pour le niveau inférieur. Pour ce faire, des ensembles de buts sont décrits et doivent permettre de produire ou déployer des plans pour les atteindre. Cela peut consister, par exemple, en un ensemble de contraintes que l'application doit absolument respecter. Le but de ce niveau est alors de trouver les solutions permettant de respecter ces contraintes même lorsqu'une perturbation est détectée au niveau *component control*. Comme dans le modèle de la robotique, il met en œuvre les mécanismes les plus complexes. Dans [121], il est réalisé à l'aide de Labeled Transition System Analyzer (LTSA) que nous avons présenté en SECTION 2.5.2.

3.2.4 Discussion

Ces architectures sur trois niveaux, qu'elles soient dédiées à la robotique ou au self-management, organisent les processus selon leur complexité et leur représentation interne du monde. Cette approche permet aux traitements les plus rapides de ne pas être dépendants de la dynamique des traitements plus lents. Les traitements devant être réalisés avec des temps de réponse les plus faibles sont placés au plus près des capteurs/actionneurs. Ce modèle architectural permet donc de prendre en

compte les diverses dynamiques de l'environnement. Cela est d'autant plus vrai que l'organisation sur trois niveaux n'est pas incompatible avec l'approche logicielle de décomposition comportementale, chaque niveau peut se baser sur cette dernière.

Contrairement à la décomposition comportementale classique, la hiérarchie entre les niveaux, permet au niveau N+1 de contrôler ou de déployer au niveau N des comportements. Grâce à ce contrôle, il est possible de garantir la cohérence des comportements au niveau N sans que cela n'entrave le bon fonctionnement des comportements déjà déployés. Cela permet d'obtenir un système dont la continuité de service est maximale. Dans la suite de ce chapitre, afin de ne plus reposer sur les appellations des niveaux propres aux différentes architectures étudiées précédemment, nous appelerons le premier niveau, équivalent au contrôleur de l'architecture 3T ou au « component control » de l'architecture pour le self-management, le niveau 0. L'appellation des niveaux supérieurs se fera ensuite de manière incrémentale, soit niveau 1 pour la couche équivalente au séquenceur et niveau 2 pour la couche équivalente au délibérateur. L'architecture n'impose aucune approche pour la gestion des comportements. Comme nous l'avons vu en section 2.5, des mécanismes de décision plus ou moins flexibles existent, chacun de ces mécanismes peut être déployé dans l'architecture et potentiellement dans des comportements différents.

Il est important de noter que cette architecture n'est pas incompatible avec l'utilisation de *models@runtime*. En effet, en considérant le niveau 0 comme reposant sur une plateforme d'exécution adaptative, le niveau 1 peut être vu comme travaillant sur le niveau méta. Dans ce cas, la relation entre les niveaux 0 et 1 se ferait à travers le mécanisme de synchronisation entre l'application s'exécutant et son modèle abstrait. Les traitements proposés par le niveau 1 seraient alors réalisés sur et en fonction du modèle abstrait de l'application. En ce basant sur une connexion lâche entre le modèle et l'application, l'indépendance des niveaux serait alors conservée. Chaque niveau poursuit sa propre dynamique et le niveau 1 permet bien le contrôle du niveau 0. Les interactions entre les niveaux sont alors le mécanisme de synchronisation entre modèle et application s'exécutant.

D'autre part, la hiérarchie entre les niveaux peut faciliter le déploiement de nouveaux comportements, grâce à l'abstraction offerte par les niveaux supérieurs, il devient plus facile de déployer dans les niveaux inférieurs des comportements cohérents, que ce soit lors de phase de conception ou de phase de déploiement. En effet, généralement, plus les niveaux mettent en œuvre des mécanismes complexes, plus ils offrent un niveau d'abstraction élevé. Cette abstraction des tâches les plus hautes, facilite les raisonnements et offre une interface de haut niveau permettant des raisonnements « *top-down* ». Ce modèle architectural, grâce aux communications entre les niveaux permet de faire des spécifications « *top-down* » pour des buts locaux. Pour rappel, il n'est pas possible d'anticiper toutes les évolutions de l'environnement, mais il est possible de spécifier, localement, un ensemble d'objectifs que l'on souhaite voir atteints par le système. Une fois ces objectifs locaux définis, le système peut alors évoluer à l'exécution dans une approche « *bottom-up* », c'est-à-dire que le système évolue, se construit pour atteindre ces objectifs en fonction des opportunités offertes par son environnement comme cela est proposé dans le cadre de l'émergence contrôlée.

Nous proposons donc dans la suite de ce chapitre de nous baser sur ce modèle architecturale et de l'étendre afin de l'adapter au cadre de l'informatique ambiante ainsi qu'une organisation des divers mécanismes de décisions existants dans les différents niveaux.

3.3 Une architecture et organisation 4 couches pour l'informatique ambiante

Dans le cadre de l'informatique ambiante, nous souhaitons adapter cette architecture. Si naturellement elle offre une bonne gestion de la multi-dynamicité du contexte d'une application, quelques évolutions sont nécessaires pour permettre la prise en compte des contraintes que nous avons présentées en introduction. En voici les principales :

- Nous avons vu, entre autre, qu'un système ambiant doit prendre en compte trois axes de variabilité et d'imprévisibilité. Cela a pour conséquence que les mécanismes de perceptions, les capteurs, ne peuvent être connus à l'avance et sont fortement variables. Il en va de même pour les actionneurs.

⇒ *Le niveau 0 doit donc reposer sur une infrastructure logicielle permettant la découverte dynamique des entités la composant. Il doit également être capable de gérer les apparitions ou disparitions des entités logicielles composant l'infrastructure.*

- D'autre part, nous ne pouvons pas connaître à l'avance, et ce, plus particulièrement dans les approches les plus flexibles, l'ensemble des adaptations à mettre en œuvre.

⇒ *Différents types de mécanismes de décision doivent pouvoir être intégrés dans l'architecture. Le mécanisme d'adaptation doit être capable de gérer des conflits entre adaptations. Parfois, il est nécessaire de faire intervenir l'utilisateur car lui seul connaît le comportement que doit avoir le système comme cela est proposé dans les architectures observer/controler. Il faut un quatrième niveau permettant l'intervention de mécanismes complexes (apprentissage) ou de l'utilisateur. Comme cela est mis en avant dans la feuille de route pour la recherche en génie logicielle pour les systèmes auto-adaptatifs du séminaire de Dagstuhl (2011) [158] : « However, such a lifecycle does not meet the requirements of self-adaptive software that we are envisioning. A self-adaptive software system operating in a highly dynamic world must adjust its behavior automatically in response to changing environments or requirements while shifting the human role from operational to strategic. »*

- Le niveau 0 n'est pas nécessairement assez évolué ni assez réactif pour détecter les erreurs de fonctionnement du système menant à une adaptation. Ce système peut également reposer sur différentes approches et ce, plus particulièrement, lorsque les déclenchements d'adaptation ne sont pas liés uniquement au bon fonctionnement de l'application. Comme les autres processus, cette fonctionnalité d'analyse du contexte, peut suivre diverses dynamiques et reposer sur divers mécanismes comme nous l'avons vu dans la section 3.1.1.4.

⇒ *Nous proposons d'alimenter en informations contextuelles tous les niveaux de l'architecture. Tous les niveaux peuvent être à l'origine du déclenchement d'une adaptation.*

- Les mécanismes de décisions que nous avons étudiés dans l'état de l'art, offrent des temps de réponse variées et sont capables d'avoir des raisonnements plus ou moins complexes. Nous devons répartir ces différents mécanismes dans l'architecture.

⇒ *Nous étudierons ces différents mécanismes ainsi que ceux d'adaptation dans l'optique de les organiser dans l'architecture en fonction de leur représentation interne du contexte et des temps de réponse qu'ils offrent.*

Nous allons dans un premier temps présenter de manière globale le modèle architectural proposé à partir de ces différents points (SECTION 3.3.1). Ensuite, nous verrons les différents mécanismes existants ainsi que leurs caractéristiques afin de déterminer de quelle manière nous allons les répartir dans ces quatre niveaux (SECTION 3.3.2). Enfin, nous présenterons de manière détaillée les différents niveaux (SECTION 3.3.3).

3.3.1 L'architecture 4 couches

L'architecture proposée doit nous permettre de mettre en place l'ensemble des mécanismes de décision que nous avons étudié dans l'état de l'art, ce qui, comme nous venons de le voir, peut impliquer de faire intervenir l'utilisateur. Ainsi, il est possible de faire évoluer le système même s'il se trouve dans une situation qui n'avait pas été prévue et qui n'est pas connue. Dans cette optique, une des évolutions apportées aux architectures précédentes est la mise en place d'un quatrième niveau. Celui-ci est le plus éloigné de l'infrastructure et de la plateforme d'exécution. Il met en jeu des mécanismes complexes qui peuvent autoriser les interactions avec l'utilisateur mais n'offrent aucune garantie quant à l'obtention d'une solution ou encore quant à ces temps de réponse.

L'architecture repose tout d'abord sur une infrastructure logicielle. Cette infrastructure doit permettre la prise en compte des dispositifs hétérogènes et volatiles qui la composent. Il faut donc qu'elle offre la capacité de découvrir dynamiquement de nouveaux dispositifs et doit avertir en cas d'apparition ou de disparition de l'un d'entre eux. Au dessus de cette infrastructure, viennent se construire les applications ubiquitaires comme des compositions d'entités logicielles de l'infrastructure. Ce niveau applicatif est inclus dans notre architecture comme celui se trouvant au plus prêt de l'infrastructure et permettant les interactions entre capteurs et actionneurs (niveau 0) .

L'architecture que nous proposons repose donc sur 4 niveaux (FIGURE 3.10). Les trois premiers (en terme de proximité avec l'infrastructure), peuvent être conçus selon une décomposition comportementale. Ils peuvent se composer de plusieurs comportements et la hiérarchie entre ces comportements (de type concurrente) est basée sur leurs temps de réponse comme présenté en FIGURE 3.5. Comme dans les modèles architecturaux sur trois niveaux, le niveau N repose sur les mécanismes offerts par le niveau N-1 et peut agir sur la couche juste en dessous. Il a la possibilité de le contrôler ou de l'adapter afin qu'il vérifie une pertinence logique. Chaque niveau poursuit sa propre dynamique avec ses abstractions, et par conséquent, son niveau de complexité en rapport avec le nombre de couches intermédiaires entre le dit niveau et l'infrastructure. Les temps de réponse sont d'autant plus importants que cette complexité est élevée. Plus le niveau est proche de l'architecture, plus il offre une dynamique au plus proche de celle de l'environnement. **L'approche hiérarchique doit permettre au niveau N+1 de garantir la pertinence logique du niveau N ; l'indépendance du niveau N a pour objectif de permettre de garantir sa pertinence temporelle.** Enfin, chaque niveau de l'architecture sera alimenté en informations contextuelles. Nous ne proposons dans cette thèse aucune hypothèse ni recommandation sur l'utilisation d'un framework pour la perception du contexte. Dans cette architecture, le déclenchement d'une adaptation peut se faire via chaque niveau. En effet, puisque chacun d'entre eux dispose d'un mécanisme de perception du contexte, ils peuvent tous détecter un changement de situation qui sera à l'origine d'une adaptation. Par contre, dans les niveaux les plus hauts, ce déclenchement de l'adaptation se fera en propageant les modifications dans les niveaux inférieurs.

Le niveau 0 garantit la mise en œuvre des routines du système, c'est-à-dire le comportement cou-

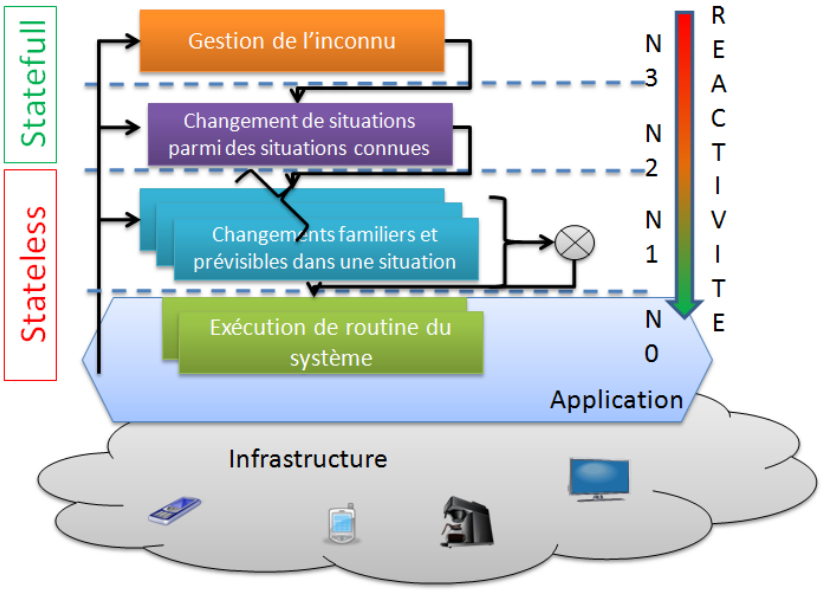


FIGURE 3.10 – Une architecture sur 4 niveaux.

rant de l'application pour la situation courante. Il représente une configuration du système. Il dispose d'une vision à court terme du comportement du système. En effet, une configuration n'est valable que pour une situation donnée et permet de réagir à des changements connus et prévus pour cette situation. Ce niveau est appelé à être modifié lorsqu'il est constaté une erreur, un dysfonctionnement, dans le comportement de routine. Il peut également être sujet à modification lorsqu'un dispositif disparaît ou apparaît dans son infrastructure logicielle. Enfin, il évolue lorsque les comportements déployés ne sont plus consistants avec la situation courante.

Le niveau supérieur permet de réagir aux changements connus mais dont l'occurrence ne peut pas être nécessairement prévue à l'avance. Par exemple, le système peut utiliser un type de dispositif particulier dans une situation mais sa présence, n'est pas certaine. Il s'agit également d'une vision à court terme du comportement du système qui n'est valable que pour une situation donnée. A partir de la définition d'un ensemble de configurations pertinentes pour une situation, il a pour objectif de déployer la plus adaptée. Le niveau 1 est appelé à être modifié, lorsqu'il perturbe la pertinence logique du système, c'est-à-dire lorsque les comportements déployés ne sont pas adaptés à la situation courante.

Le niveau 2, permet de réagir aux changements qui ne sont pas familiers à la situation courante. Il permet de faire passer le système dans une configuration adaptée à une nouvelle situation. Ce niveau à une vision à moyen terme du comportement du système, et reste valide tant que l'on subsiste dans des situations prévues. Pour ce faire, il déploie dans le niveau inférieur l'ensemble des configurations pertinentes dans la nouvelle situation. Il lui est possible d'entraver la pertinence logique du système lorsqu'il active ou déploie dans le niveau inférieur des règles inadaptées.

Enfin, le niveau 3 permet de réagir aux changements considérés comme anormaux, c'est-à-dire menant à une situation qui n'a pas été prévue et qui est non-connue. Pour gérer cela, bien souvent le meilleur moyen est de faire intervenir l'utilisateur. En effet, lui seul connaît le comportement et

l'objectif qu'il souhaite voir le système atteindre. Ce niveau dispose de la vision à plus long terme du comportement du système. Avec l'expérience, il doit être de moins en moins sollicité. Comme le précédent, les risques associés à ce niveau sont de déployer des règles qui ne sont pas adaptées au contexte. Un autre risque à envisager est la possibilité qu'il ne trouve pas de solution.

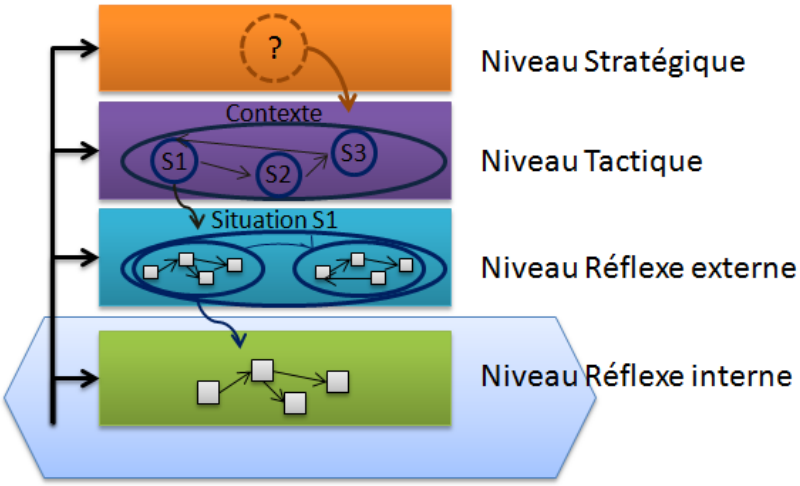


FIGURE 3.11 – Objectifs des différents niveaux.

Il est important de noter que tous les niveaux ne doivent pas être nécessairement présents à l'exception du niveau 0. D'autre part, comme les boucles de contrôle classique de l'autonomic computing, la granularité de cette architecture dépend des entités auxquelles elle est attachée. Il peut s'agir d'un composant composite comme d'un système global. Comme les architectures Observer/Controler, elles peuvent être utilisées dans le système de manière décentralisée, hiérarchique ou centralisée. La FIGURE 3.12 présente une utilisation hiérarchique de l'architecture.

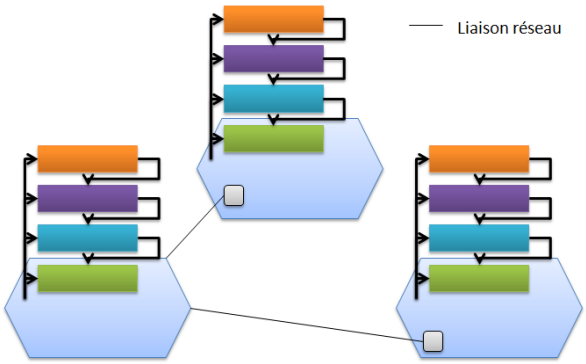


FIGURE 3.12 – Utilisation de l'architecture de manière hiérarchique.

Tout type de modèle du contexte ou de mécanisme de décision est susceptible d'être utilisé dans cette architecture et doit y trouver sa place. Nous allons maintenant étudier les différentes approches de traitement et d'exploitation du contexte existantes. Nous verrons, ensuite, comment les répartir afin qu'elles autorisent la réalisation par chaque niveau de son objectif mais aussi de manière à ce que les

mécanismes les plus complexes et coûteux en temps se trouvent dans les niveaux les plus complexes qui sont les plus éloignés de l'infrastructure.

3.3.2 Les différentes approches de traitement et exploitation du contexte à répartir dans les 4 niveaux de l'architecture

Que ce soit dans la décomposition comportementale (SECTION 3.2.1) ou dans le décomposition fonctionnelle classique des intergiciels sensibles au contexte, quatre grandes fonctionnalités sont généralement présentes : la perception, l'analyse des données contextuelles, la décision et la réaction. Les étapes de perception, analyse et décision peuvent être regroupées dans une grande thématique de traitement du contexte. La réaction peut être assimilée à l'exploitation du contexte. Diverses approches de traitement et d'exploitation du contexte existent ; nous allons voir comment les répartir dans l'architecture. Comme en robotique nous ordonnerons les niveaux en fonction de leur gestion de l'état du monde, mais aussi en fonction de leurs temps de réponse.

3.3.2.1 Deux approches pour traiter le contexte

Deux grandes approches de traitement du contexte peuvent être identifiées dans la littérature en fonction de leur gestion de l'état de l'environnement. Les approches **logiques** et **physiques**.

L'approche logique est la plus utilisée. Elle consiste à collecter un maximum d'observables dans une base de connaissances, ce qui permet, par la suite, la mise en place de raisonnements complexes d'analyse et d'abstraction des données. Classiquement, les informations contextuelles sont ensuite obtenues dans une entité centralisée. Souvent, ces serveurs de contexte se trouvent sur des plateformes qui ne sont pas contraintes par leur consommation énergétique. Même si les informations contextuelles sont décentralisées, une caractéristique des approches logiques est qu'elles conservent un état du contexte. Ce type d'approche est privilégiée dans les architectures verticales. Les mécanismes mis en place dans ces approches peuvent être complexes. Nous pouvons distinguer les mécanismes pour lesquels les temps de calcul sont finis, comme les ontologies, de ceux pour lesquels les temps de calcul ne le sont pas. Il s'agit par exemple d'approches faisant intervenir l'utilisateur comme les algorithmes évolutionnaires interactifs ou des mécanismes d'apprentissage.

L'approche physique , plus simple et réactive, est apparue avec [41] pour les systèmes « *location-aware* ». L'objectif est alors de prendre en compte les informations contextuelles accessibles par les capteurs se trouvant à proximité de l'entité de référence. Dans cette approche, les données ne sont plus stockées dans une base de connaissance comme pour l'approche logique et le système ne conserve aucune représentation interne de son contexte. Ce type d'approche est plus naturellement décentralisée mais permet difficilement de réaliser des traitements complexes. L'efficacité de l'approche physique est meilleur puisque seules les informations provenant d'un nombre limité de sources proches de l'entité sont collectées et considérées. En l'absence d'état et de représentation du contexte, la perception et la décision se font principalement en mode *push*, c'est-à-dire aux fils des notifications de changements de l'environnement. A l'origine, les zones définissant la portée d'un contexte étaient quasiment basées uniquement sur la localisation. V. Hourdin dans sa thèse [86], étend cette notion et la généralise à tout type de contexte.

Un même système peut combiner approches logiques et physiques. Par exemple dans le cadre d'une décomposition comportementale, chaque comportement peut utiliser l'approche qui lui

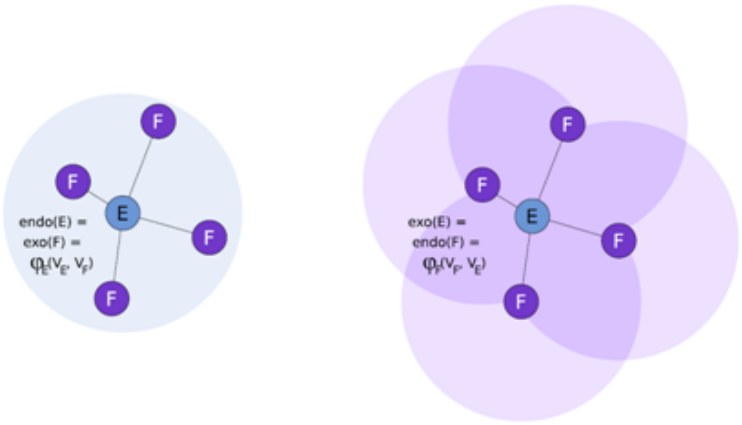


FIGURE 3.13 – Approche physique : les zones contextuelles.

convient. S'ils utilisent une approche logiques alors ils ne correspondent plus exactement à la définition d'un comportement réactif tel que posée par R. Brooks puisqu'avec un état et une représentation interne de l'environnement. Pour garantir la cohérence du système, si la hiérarchie est dictée par les temps de réponse, c'est au mécanisme d'exploitation du contexte de résoudre les potentiels conflits. Dans le cadre des architectures *3-layers*, la hiérarchie entre les niveaux permet de gérer cette cohérence. Aux deux approches peuvent être combinés différents types d'adaptation.

3.3.2.2 Combiner adaptation compositionnelle et paramétrée

Nous avons vu dans l'état de l'art (SECTION 2.2), qu'il existe deux grandes approches d'adaptation : paramétrée et compositionnelle. L'adaptation paramétrée consiste à modifier la valeur de variables afin de changer le comportement d'un logiciel. Ce type d'adaptation réduit très fortement les temps d'adaptation et limite les interruptions de l'application. D'autre part, il est aisé d'associer aux paramètres des mécanismes de prise en compte du contexte indépendants les uns des autres et ainsi de faire évoluer l'application avec plusieurs dynamiques (FIGURE 3.7). Par contre, les adaptations sont limitées aux stratégies existantes. Ce type d'adaptation n'est pas adapté à la prise en compte de l'imprévisibilité de l'infrastructure logicielle d'un système ambiant. Elle ne permet pas de prendre en compte de nouvelles entités logicielles.

A l'inverse l'adaptation compositionnelle permet de modifier la structure d'un programme, d'ajouter, retirer, échanger des algorithmes. Avec ce type d'adaptation, le temps d'adaptation est plus conséquent et les interruptions de l'application plus importantes. Par contre, elle permet d'intégrer dans le système de nouveaux types de stratégie sans autre limitation que le maintien de la cohérence du système. Ce type d'adaptation est particulièrement adapté à la prise en compte des évolutions de l'infrastructure logicielle du système.

Adaptations compositionnelles et paramétrées peuvent être combinées de manière à tirer bénéfice des deux approches. En particulier, l'adaptation compositionnelle permet de mettre en place de nouveaux paramètres d'adaptation. La première peut alors, par exemple, être plus spécifiquement dédiée aux adaptations dues à des variations de l'infrastructure logicielle, tandis que la seconde peut être liée à la prise en compte de l'environnement et dédiée à des adaptations qui doivent être réalisés le plus rapidement possible (FIGURE 3.14), nous pourrons parler d'adaptations critiques.

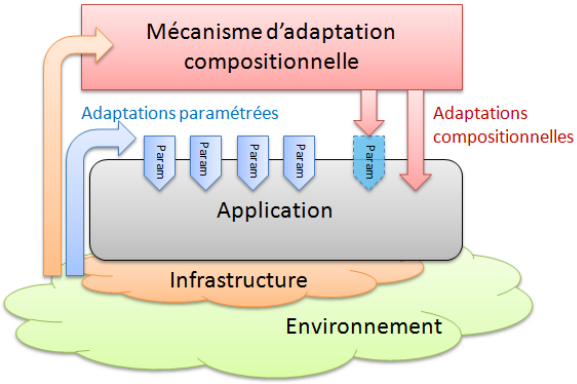


FIGURE 3.14 – Un exemple d'utilisation concerté de l'adaptation compositionnelle et paramétrée

Les différentes approches de traitement et d'exploitation du contexte que nous venons de voir peuvent être combinées les unes avec les autres.

3.3.2.3 Synthèse

Nous avons vu précédemment que nous souhaitons organiser ces différentes combinaisons dans notre architecture en fonction de deux critères principaux : (1) leurs temps de réponse et (2) leur utilisation d'une représentation interne de l'environnement. Un autre critère : l'extensibilité de l'approche, pourra entrer un compte. Pour les combinaisons proposant un traitement du contexte logique, nous pouvons considérer que, quelque soit l'approche d'adaptation choisie, ses temps de réponse sont négligeables en comparaisons de ceux des mécanismes de traitement du contexte. Par contre, nous avons vu que les mécanismes utilisés dans les approches logiques ne réalisent pas toujours leurs traitements dans des temps finis. A l'inverse, dans les approches physiques, le type d'adaptation joue un rôle important quant au temps de réponse. Nous pouvons donc distinguer quatre types de comportements. Les comportements reposant sur un traitement du contexte logique sont les plus lents mais offrent des capacités de raisonnements les plus poussées. Les comportements reposant sur un traitement du contexte physique sont les plus rapides, en particulier ceux réalisant des adaptations paramétrées sont plus rapides que ceux réalisant une adaptation compositionnelle. Le tableau 3.1 présente les principales caractéristiques des différentes approches.

Nous allons maintenant présenter de quelle manière sont organisées ces approches dans les différents niveaux de l'architecture. Pour ce faire, nous détaillerons chaque niveau et identifierons pour chacun d'entre eux l'approche retenue.

3.3.3 Organisation des mécanismes d'exploitation du contexte et d'adaptation dans les 4 niveaux

Les quatre types de combinaisons <traitement-exploitation du contexte> sont donc organisés dans les quatre couches de l'architecture en fonction de leurs caractéristiques présentées dans le tableau récapitulatif 3.1. Les approches les plus lentes mais permettant des raisonnements complexes seront dans les niveaux les plus éloignés de l'application. Nous allons maintenant détailler chacun de ces niveaux. A chaque description, nous associerons un tableau récapitulatif de ces caractéristiques.

TABLE 3.1 – Caractéristiques des différentes combinaisons traitement-exploitation du contexte.

Combinaison	**Capacités de raisonnement**	**Extensibilité de l'adaptation**	**Temps de réponse**
Logique en temps non-fini	Maximales (avec état)	Oui (si adaptation compositionnelle)	Important sans garantie d'obtenir un résultat
Logique en temps fini	Très bonnes (avec état)	Oui (si adaptation compositionnelle)	Important mais finis
Physique-compositionnelle	Limitées (sans état)	Oui	Faibles et finis mais plus important que dans l'approche physique-paramétrée
Physique-paramétrée	Limitées (sans état)	Limitée aux paramètres définis	Les plus faibles et bornés

3.3.3.1 Niveau 0, réflexe interne : traitement physique et adaptation paramétrée

Le premier niveau, que nous appellerons réflexe interne, équivalent au niveau « contrôleur » de l'architecture 3T, a pour objectif de mettre en relation capteurs et actionneurs et ainsi de créer des applications. Il s'agit donc du niveau permettant la composition des entités logicielles de l'infrastructure. Il devra donc être en mesure de prendre en compte dynamiquement les apparitions/-disparitions de ces entités logicielles. Il se composera donc d'une plateforme d'exécution permettant de réaliser des adaptations compositionnelle. La plateforme doit permettre la création d'applications modulaires. Nous avons donc vu en SECTION 2.2.2.2 que les approches orientées composants sont particulièrement adaptés à cela. Par conséquent, ce niveau repose sur une plateforme d'exécution à base de composants qui offre des capacités d'introspection et doit permettre l'ajout et le retrait de composants/liaisons. D'autre part, la plateforme doit disposer d'un mécanisme notifiant de tout changement dans l'assemblage.

Les processus de traitements des informations contextuelles et d'adaptation se trouvant à ce niveau sont donc internalisés dans l'application et prennent la forme d'assemblages de composants. Les temps de réponse à ce niveau devant être les plus faibles possibles, les traitements du contexte seront physiques et l'adaptation paramétrée. Les paramètres seront alors le moyen de modifier le comportement des composants instanciés. D'autre part, des composants spécifiques pourront être déployés afin de permettre de filtrer les interactions entre composants, comme cela est proposé dans [86]. De cette manière, des comportements (SECTION 3.2.1), appelés réflexes internes, pourront être déployés par le niveau supérieur dans l'application sous la forme d'assemblage de composants.

3.3.3.2 Niveau 1, réflexe externe : traitement physique et adaptation compositionnelle

Le second niveau, que nous appellerons réflexe externe, prend la forme d'un ensemble de comportements dont l'objectif est de réaliser des adaptations compositionnelles du niveau inférieur. Ces comportements utilisent un traitement des informations contextuelles physique et une adaptation compositionnelle. Ils pourront déployer, retirer ou modifier des réflexes internes. Cela consiste en l'ajout ou retrait de composants ou liaisons. Ce niveau permet donc de répondre à une problématique

TABLE 3.2 – Caractéristiques du niveau réflexe interne.

Objectif :	Garantir l'exécution de routine du système ambiant dans une situation donnée.
Mise en œuvre :	Approche physique et adaptation paramétrée. Plateforme d'exécution adaptative et implémentation de comportements dans l'application par assemblage de composants.
Impact par rapport au comportement global du système :	Impact à court terme : ce niveau n'est valide que le temps d'une situation et pour une infrastructure qui ne change pas.
Apport concernant la pertinence temporelle :	• Permet de réagir rapidement aux changements de contexte considérés. • La durée de l'adaptation est courte.
Ce qui peut entraver la pertinence logique :	• Erreurs, dysfonctionnements, dans la routine. • Disparition de dispositifs. • Les comportements déployés ne sont plus adaptés au contexte.

du niveau inférieur, à savoir le manque d'extensibilité des approches paramétrées.

Tous les comportements sont au même niveau et sans hiérarchie. Nous avons vu qu'un système ambiant doit respecter plusieurs dynamiques dont une obligatoirement et de la manière la plus réactive possible : celle de l'infrastructure d'une application ambiante. Chaque comportement de ce niveau doit prendre en compte les évolutions de l'infrastructure de l'application. Dans le plus simple des cas, à l'adaptation est associé un unique mécanisme de décision chargé de vérifier si l'infrastructure permet bien sa mise en œuvre.

Nous avons également vu que ce type d'adaptation doit être externalisé de l'application dans ce que l'on appel une approche transparente [50]. Parmi les niveaux externalisés, il est le plus rapide. Comme le premier niveau, celui-ci est également alimenté en informations contextuelles. De plus, il est informé des modifications intervenant au niveau inférieur. Le mécanisme d'adaptation associé à ce niveau peut reposer sur une approche de type *models@runtime*. Si tel est le cas, il utilisera les capacités d'introspection de la plateforme sous-jacente afin d'être synchronisé avec le niveau inférieur. Les adaptations décrites à ce niveau doivent respecter l'ensemble des caractéristiques que nous avons définies en SECTION 2.6. En particulier, puisque divers mécanismes de décision pourront leur être associés, y compris les plus flexibles, le mécanisme d'adaptation devra incorporer un mécanisme de gestion des conflits d'adaptation.

3.3.3.3 Niveau 2, tactique : traitement logique en temps fini

Le troisième niveau, que nous appellerons tactique, prend la forme d'un ou plusieurs comportements s'appuyant sur le niveau inférieur, dont l'objectif est de déployer, sélectionner ou désélectionner des comportements réflexes externes. Il permet de gérer la bonne cohérence de ces derniers, par exemple en les bloquant ou encore en leur indiquant une politique de mise en œuvre. Pour ce faire, ces comportements utilisent un traitement des informations contextuelles logique. Ce

TABLE 3.3 – Caractéristiques du niveau réflexe externe.

Objectif :	Réagir aux changements familiers et qui étaient prévisibles dans cette situation.
Mise en œuvre :	Approche physique et adaptation compositionnelle, permettant ajout/retrait de composants/liaisons.
Impact par rapport au comportement global du système :	Impact à court terme : ce niveau n'est valide que le temps d'une situation.
Apport concernant la pertinence temporelle :	• Permet de réagir rapidement aux changements de contexte considérés et plus particulièrement aux disparitions de dispositifs. • La durée de l'adaptation est courte.
Ce qui peut entraver la pertinence logique :	• Les comportements déployés ne sont plus adaptés au contexte.

niveau est donc également alimenté en informations contextuelles ainsi qu'en informations relatives à l'état du niveau inférieur.

Les comportements déployés dans ce niveau n'ont pas nécessairement besoin d'avoir une dynamique aussi proche que les réflexes de l'application et de l'environnement. Toutefois, ils doivent quand même offrir des temps de réponse raisonnables et finis. Le traitement logique mis en œuvre dans ce niveau doit par conséquent être fini. Nous pouvons donc associer à ce niveau une base de connaissance ou tout autre mécanisme jouant le rôle de serveur de contexte. C'est à ce niveau que nous pourrons trouver les différents mécanismes de décisions plus ou moins flexibles présentés en SECTION 2.5. Par exemple, dans le cas d'un mécanisme de décision contraint basé sur un automate à états finis, chaque état de ce dernier définirait les comportements réflexes et donc les adaptations à déployer. Dans une approche moins contrainte, utilisant par exemple un diagramme de feature, la sélection d'une feature pourrait correspondre au déploiement d'un comportement.

Enfin, ce niveau peut définir des politiques de mise en place des comportements du niveau inférieur. Par exemple, il pourra décider que la réalisation de certains comportements ne peut se faire que dans des intervalles réguliers. Une autre politique peut consister à réaliser au plus tôt les adaptations dues à des disparitions de dispositifs et à demander validation des adaptations dues à des apparitions de dispositifs. Une telle approche permettant de coller aux critères définis en SECTION 1.2.4.

3.3.3.4 Niveau 3 Stratégique : traitement logiques en temps non-finis

Le quatrième niveau, que nous appellerons stratégique, a pour objectif de permettre la réaction du système à des situations inconnues et imprévues. Ce niveau étant le plus complexe, mais aussi celui de plus haut niveau, il est difficile à décentraliser ou encore à concevoir comme un ensemble de comportements. Le niveau stratégique peut reposer sur des mécanismes complexes dont les temps de réponse ne sont ni bornés ni finis et n'offrent aucune garantie de résultat. La dynamique de ce niveau est totalement décorrélée de celle de l'environnement. Ces mécanismes peuvent se baser sur des techniques d'apprentissage ou encore mettre en jeu des utilisateurs (administrateur ou encore utilisateur final). Pour cela, comme les autres niveaux, il est alimenté en informations contextuelles.

TABLE 3.4 – Caractéristiques du niveau Tactique.

Objectif :	Réagir aux accidents non familiers à cette situation (déclencher le changement de situation.)
Mise en œuvre :	Approche logique. Déploiement ou sélection de nouveaux réflexes externes.
Impact par rapport au comportement global du système :	Impact à moyen terme : ce niveau est valide tant que l'on reste dans ce qui pouvait être prévu.
Apport concernant la pertinence temporelle :	• Temps de réponse finis.
Ce qui peut entraver la pertinence logique :	• Activer des règles inadaptées.

Il s'appuie sur le niveau tactique et a pour objectif d'ajouter de nouvelles règles aux comportements tactiques existants ou d'en déployer de nouveaux. Par exemple, dans les approches basées sur des types, ajouter de nouvelles contraintes permet de générer un nouvel espace de solutions. Dans les approches proposant de l'émergence contrôlée, il s'agit d'un moyen supplémentaire de gérer les trois axes d'imprévisibilité en modifiant la partie contrôle du mécanisme de décision. Ceci aurait pour conséquence la modification en cascade des niveaux inférieurs, comme par exemple, le déploiement de nouveaux comportements dans les niveaux inférieurs.

TABLE 3.5 – Caractéristiques du niveau Stratégique.

Objectif :	Réagir aux changements considérés comme anormaux et non prévus. S'adapter à des situations non prévues et non connues.
Mise en œuvre :	Approche logique. Mise à jour du niveau tactique (ajout/retrait/modifications de règles dans un format dépendant des modèles sous-jacents).
Impact par rapport au comportement global du système :	Impact à long terme : vision globale du système et de ces objectifs.
Apport concernant la pertinence temporelle :	• Temps de réponse inadaptés.
Ce qui peut entraver la pertinence logique :	• Déployer des règles inadaptées. • Ne pas trouver de solution.

3.4 Synthèse

L'architecture présentée, propose un cadre de travail pour la conception de systèmes ambiants et donc sensibles au contexte (FIGURE 3.15). Elle propose d'organiser les différentes approches de traitement du contexte et d'adaptation que nous trouvons dans la littérature de manière à répondre au mieux à un certain nombre des contraintes définies dans l'introduction.

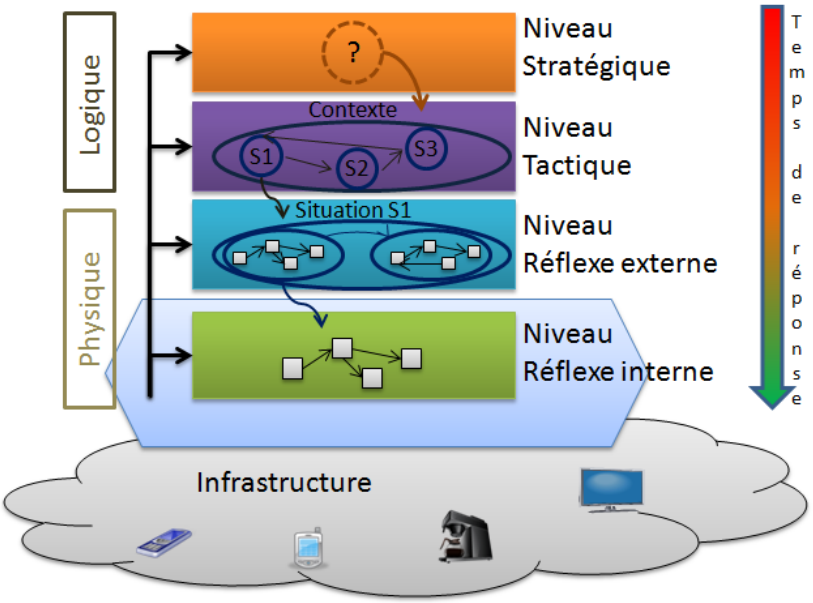


FIGURE 3.15 – Modèle architectural sur quatre niveaux.

– Grâce à l'indépendance des couches et à la hiérarchie établie entre elles, basée sur leurs temps de réponse ainsi que leur gestion de l'état de l'environnement, elle permet de respecter les diverses dynamiques d'évolution de l'environnement. Ainsi, elles garantissent la pertinence temporelle de l'adaptation tout en autorisant le maintient de la pertinence logique du système.
– De plus, les traitements étant indépendants les uns des autres, lorsqu'un processus de prise en compte du contexte complexe (par exemple dans les niveaux tactiques ou stratégiques) est lancé, les processus plus simples (réflexes) restent toujours en fonctionnement et l'application également. La continuité de service est maximale.
– La conception du système peut être entièrement *top-down*, puisque les niveaux les plus abstraits permettent de déployer les plus concrets, ou relever d'un compromis entre conception *top-down* et *bottom-up* lorsqu'une certaine émergence est réalisée par les niveaux réflexes puis contrôlée par les niveaux tactiques et stratégiques.
– L'architecture est suffisamment générique pour permettre l'utilisation en son sein des différentes approches de décision et d'adaptation que nous avons vu dans l'état de l'art. Par conséquent, le mécanisme d'adaptation qui assurera le management tardif des comportements réflexes doit garantir une bonne indépendance entre les adaptations, être extensible et par conséquent résoudre les potentiels conflits d'adaptation. Pour cela, il doit offrir une bonne modularité et reposer sur le principe de la séparation des préoccupations. La composition de ces entités doit pouvoir être totalement maîtrisée à travers l'expression de contraintes entre les entités d'adaptation ou la plus flexible.

Nous allons maintenant proposer un mécanisme d'adaptation susceptible d'être couplée à notre architecture. Le mécanisme d'adaptation proposé respectera les caractéristiques que nous avons définies dans l'état de l'art, et devra pouvoir être associé à la plupart des types de mécanismes de décision, du plus contraint au plus flexible. Nous étudierons quelles sont les contraintes que cela implique. Enfin, nous conserverons à l'esprit la préoccupation transversale de conservation de temps de réponse faibles.

CHAPITRE 4

Mécanisme d'adaptation : la séparation des préoccupations poussée à son paroxysme

« Non, ne nous trompons pas ! C'est un exercice continuel d'adaptation et d'évolution qui nous attend et menace notre équilibre de façon constante. »

Daniel Desbiens.

Sommaire

CE CHAPITRE présente un mécanisme d'adaptation pouvant être intégré dans notre architecture à quatre niveaux. Il peut être utilisé par des mécanismes de décision plus ou moins flexibles. L'approche proposée est construite à partir des constats que nous avons tirés de l'analyse de l'état de

l'art et respecte donc les contraintes que nous en avons extrait.

Dans un premier temps, nous reviendrons sur ces contraintes imposées aux mécanismes d'adaptation dans le cadre de l'informatique ambiante. Nous verrons ensuite l'approche proposée et de quelle manière elle respecte ces contraintes. Afin de pouvoir trouver sa place dans l'architecture sur 4 niveaux comme un mécanisme permettant l'adaptation du niveau réflexe interne, la solution proposée devra pouvoir être utilisée, dans les couches supérieures, par des mécanismes de décision rigides comme flexibles. L'ensemble des adaptations est donc extensible et elles sont indépendantes les unes des autres. Pour ce faire, l'approche proposée, de type « models@runtime » et permettant de réaliser des adaptations compositionnelles, doit permettre de définir l'adaptation comme une préoccupation transverse homogène et hétérogène tout en garantissant un ensemble de propriétés logiques que nous étudierons dans ce chapitre. Enfin, nous présenterons une mise en œuvre de cette approche que nous évaluerons en termes de temps réponse.

4.1 Retour sur les contraintes

L'analyse de l'état de l'art nous a mené à la définition d'un ensemble de contraintes que doivent respecter les mécanismes d'adaptation en informatique ambiante. L'ensemble de celles-ci est résumé dans le tableau 4.1.

TABLE 4.1 – Contraintes pour les mécanismes d'adaptation en informatique ambiante.

Adaptation dynamique	π_1
Adaptation compositionnelle	π_2
Adaptation portée sur des modèles à l'exécution	π_3
Adaptation comme préoccupation transverse homogène et hétérogène	π_4
Opération d'adaptation déterministe	π_5
Ensemble des adaptations extensibles	π_6
Temps de réponse maîtrisés et adaptés	π_7

Le contexte peut évoluer fréquemment ; s'y adapter peut être à l'origine de nombreuses évolutions du système. Il n'est donc pas envisageable d'arrêter et de redéployer le système à chaque fois qu'une évolution est nécessaire. Nous avons donc, tout d'abord, identifié le besoin de disposer d'un mécanisme permettant de réaliser des adaptations **dynamiques**. De plus, ces adaptations doivent être à minima **compositionnelles**. De cette manière, il sera possible d'intégrer, à l'exécution, de nouvelles briques logicielles qui peuvent être des entités se trouvant dans l'infrastructure logicielle de l'application et qui ne sont pas nécessairement modifiables. D'autre part, puisqu'elles peuvent être mobiles, l'hypothèse de leur existence à un instant donné ne peut être ni émise ni même anticipée. Ce type d'adaptation nécessitant d'interrompre le fonctionnement d'au moins une partie de l'application, il est important de minimiser ce phénomène. L'adaptation doit donc perturber le moins possible le bon fonctionnement du système et le résultat de l'adaptation doit être analysé avant son application. Pour ce faire, nous utiliserons des **modèles de l'application à l'exécution**. Enfin, l'adaptation doit être décrite et gérée comme une préoccupation transverse dans l'optique de réduire l'enchevêtrement du code pour augmenter la réutilisabilité et l'indépendance des adaptations. Ces **préoccupations transverses peuvent être homogènes ou hétérogènes**. Ces entités, qui sont alors indépendantes, doivent être composées et le mécanisme d'adaptation doit garantir le **déterminisme** du résultat. Lorsque le

mécanisme d'adaptation est associé à un mécanisme de décision flexible, il doit être possible d'ajouter des entités d'adaptation à l'exécution sans se préoccuper des autres adaptations présentes, c'est-à-dire sans spécifier de quelle manière les composer les unes avec les autres. L'ensemble des adaptations doit donc être **extensible** et les adaptations doivent être **indépendantes** les unes des autres. De la même manière, elles doivent pouvoir être déclenchées indépendamment les unes des autres. Pour cela, le mécanisme d'adaptation doit inclure un système de gestion des conflits entre adaptations. Parallèlement à tous cela, le mécanisme d'adaptation doit remplir tous ces critères tout en garantissant des **temps de réponse maîtrisés et adaptés**.

4.2 Approche proposée : des cascades d'aspects

Nous définissons, à partir de ces contraintes, notre approche pour la définition d'un mécanisme d'adaptation. Le mécanisme proposé permet, à l'exécution, de calculer des adaptations et de produire des scripts de reconfiguration à partir d'un ensemble d'entités d'adaptation et d'une application de base. L'approche choisie repose sur des plateformes d'exécution à base de composants afin de faciliter l'adaptation compositionnelle (SECTION 4.2.1). Par conséquent, les scripts de reconfiguration sont des ensembles d'instructions élémentaires de reconfiguration comme ajouter un composant ou une liaison entre deux composants. D'autre part, le calcul des modifications à porter dans l'application est fait à partir d'une représentation abstraite de l'application s'exécutant, ce qui permet de vérifier des propriétés sur le résultat de l'adaptation (SECTION 4.4.3).

Les entités décrivant les adaptations à réaliser (SECTION 4.2.2) données en entrée au mécanisme d'adaptation prennent la forme de comportements (SECTION 4.2.2.2). Ces derniers, dans l'optique de faciliter l'expression des préoccupations transverses homogènes et hétérogènes, s'inspirent des approches orientées features et aspects. Pour ce faire, ils reposent sur une décomposition fonctionnelle dont les composantes sont exprimées sous la forme d'aspects (SECTION 4.2.2.1). Grâce aux propriétés de l'opération de tissage de ces comportements (en particulier la propriété de symétrie), l'ensemble de ceux-ci est extensible et il n'est pas nécessaire d'exprimer de dépendances entre eux (SECTION 4.2.3). Ces comportements peuvent donc être composés et organisés de manière rigide et contrainte ou de manière opportune en fonction de l'infrastructure logicielle sous-jacente. La FIGURE 4.1 présente de manière globale l'approche proposée.

Enfin, l'approche est mise-en-œuvre comme une extension des Aspects d'Assemblages [114] (SECTION 4.3 et 4.4), et nous garantit des propriétés temporelles du mécanisme proposé (SECTION 4.5).

4.2.1 Adaptation structurelle et dynamique

Nous avons vu dans l'état de l'art (SECTION 2.2.2) que pour réaliser des adaptations compositionnelles, l'application ciblée par l'adaptation doit être modulaire et que les modules doivent pouvoir être facilement composés. Les approches orientées services et composants sont de bonnes candidates pour la mise en œuvre de ces applications grâce au faible couplage entre les entités logicielles. Dans les approches à composants, le couplage faible entre composants et le contrôle de leur environnement d'exécution facilite leur remplacement dynamique tandis que les approches orientées services offrent plus particulièrement des solutions à la gestion de l'hétérogénéité et à la distribution des entités logicielles (SECTION 2.2.2). L'interopérabilité sur les protocoles de communications, le matériel ou

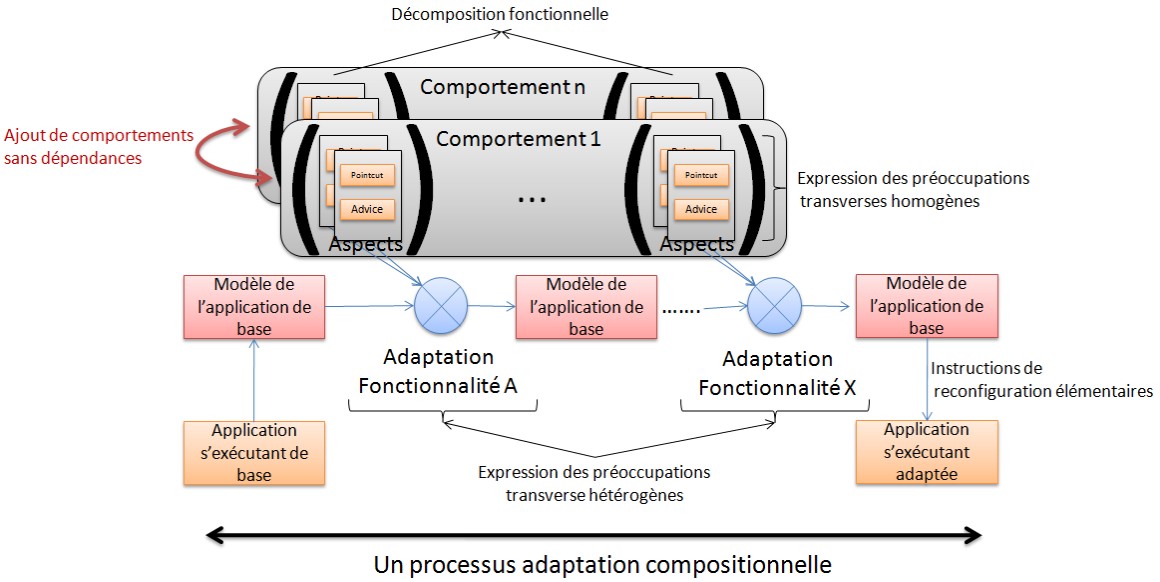


FIGURE 4.1 – Vision globale de l'approche proposée

encore les langages de programmation peut être prise en charge par les services Web. Le support de notre mécanisme d'adaptation sera donc une plateforme adaptative à base de composants. Nous répondons ainsi à la première contrainte (π_1). D'autre part, nous avons vu que les applications d'informatique ambiante sont créées en faisant interagir entre eux des dispositifs se trouvant dans leur infrastructure logicielle. Ces dispositifs peuvent être hétérogènes et distribués, les approches à services offrent alors l'interopérabilité nécessaire pour permettre les interactions entre dispositifs. Ainsi, plus que les plateformes adaptatives à base de composants, nous privilégierons les plateformes multi-paradigmes couplant à la fois composants et services, c'est-à-dire permettant d'orchestrer des services par assemblage de composants.

Puisque les parties logicielles déployées sur les dispositifs de l'infrastructure ne sont pas nécessairement modifiables, des composants boites noires représentant ces parties logicielles peuvent être utilisées afin de faire interagir ces dispositifs entre eux. Par exemple, une apparition dans l'infrastructure logicielle est intégrée dans l'application par la création et l'instanciation d'un représentant vers le nouveau dispositif. Ces composants doivent donc bien être vus comme des boites noires et le mécanisme d'adaptation associé ne sera donc pas intrusif.

L'objectif du mécanisme d'adaptation sera de produire un ensemble d'instructions élémentaires d'adaptation permettant de modifier l'architecture du système, à savoir : ajouter ou retirer un composant, une liaison ou modifier une propriété d'un composant, ce qui nous permet de répondre à la seconde contrainte (π_2). Le mécanisme d'adaptation sera externalisé de l'application de manière à ce que l'application soit interrompue uniquement lors de la mise en œuvre des instructions élémentaires. Dans cette optique, et afin de vérifier la cohérence de ces adaptations, les adaptations seront portées à l'exécution sur un modèle de l'application avant d'être portées dans l'application s'exécutant. Nous vérifions ainsi la contrainte (π_3). L'adaptation étant une préoccupation transverse, nous utiliserons, pour produire ces instructions, des techniques permettant de gérer la séparation des préoccupations transverses.

4.2.2 Décomposer et réutiliser pour mieux gérer la variabilité

Nous avons vu dans l'état de l'art (SECTION 2.4.2) que de nombreux travaux proposent d'utiliser des approches orientées aspects dans l'optique de réaliser des adaptations dynamiques. Les aspects, en aplatissant les structures hiérarchiques que l'on retrouve par exemple dans les approches orientées objets classiques, permettent difficilement d'exprimer des préoccupations transverses hétérogènes [3]. Afin d'autoriser l'expression de l'adaptation comme une préoccupation transverse homogène ou hétérogène, nous proposons de nous inspirer des approches orientées aspects et features (π_4). La décomposition que ces approches autorisent, favorise la réutilisation des adaptations ainsi que leur capitalisation. Grâce à la modularisation des adaptations, il devient plus facile d'identifier les adaptations à composer pour créer diverses variantes du système.

Traditionnellement, une autre caractéristique des approches orientées aspects est de gérer les interactions entre aspects en exprimant des contraintes entre eux. Généralement, cela se fait à travers des notions de contrats ou de précédence. Ces approches pour l'adaptation sont alors appropriées pour être associées aux mécanismes de décision les plus rigides et, au contraire, peuvent difficilement être utilisées par les mécanismes de décision les plus flexibles.

Pour répondre à ces problématiques, nous définissons une adaptation comme un ensemble de fonctionnalités qui peuvent être réalisées de diverses manières, de telle sorte qu'il sera possible d'en mettre en œuvre diverses variantes. Dans cette optique, une fonctionnalité prendra la forme d'un ensemble d'aspects qui peuvent être combinés afin d'en créer une variante. La fonctionnalité pourra ainsi être dupliquée plusieurs fois dans l'application. Les aspects déployés pour une fonctionnalité sont alors autant de variantes de cette fonctionnalité et peuvent être composés. Nous appellerons une entité d'adaptation contenant un ensemble de fonctionnalités, qui sont elles-mêmes spécifiées comme des ensembles d'aspects, des *cascades d'aspects*.

4.2.2.1 Les cascades d'aspects

Le premier levier permettant aux mécanismes de décision de maîtriser les adaptations devant être réalisées, sera la capacité de déployer ou retirer des cascades et des aspects à l'exécution. Ce levier offre un médium, pour les mécanismes de décision déployés dans les différents niveaux de l'architecture, pour garantir la cohérence sémantique des adaptations déployées. Lorsqu'un mécanisme de décision basé sur une approche explicite est associé aux adaptations, il est également nécessaire de pouvoir indiquer au tisseur des relations entre les aspects. Pour ce faire, il sera possible d'exprimer un ordre entre les aspects. Au contraire, dans les cas les plus flexibles, nous disposerons ensemble tous les aspects qui se composeront les uns avec les autres en fonction de leurs capacités à être tissés, c'est-à-dire en fonction de la présence de points de jonctions vérifiant leurs points de coupes. Enfin, entre ces deux extrêmes, il sera possible de définir des fonctionnalités qui seront des ensembles d'aspects indépendants. Dans ces fonctionnalités, aucune contrainte ne sera exprimée ; par contre, il sera possible d'ordonner ces ensembles entre eux.

Les cascades d'aspects doivent permettre la mise en place de ces trois cas. Pour ce faire, nous définissons une cascade comme un ensemble ordonné d'ensembles d'aspects. Dans le cas le plus contraint, chaque ensemble appartenant à la cascade peut contenir un unique aspect, de telle manière que tous les aspects sont ordonnés les uns par rapport aux autres. Dans les approches les plus flexibles,

une cascade se composera d'un unique ensemble d'aspects. Une cascade se définit donc comme présentée dans l'EQUATION 4.1. Dans cette équation, $CAspect_i$ est la cascade d'aspect d'indice i. Elle se compose de j ensembles d'aspects. Chacun de ces aspects est identifié tout d'abord par l'indice de l'ensemble auquel il appartient (jusqu'à j) et par un indice permettant de l'identifier dans cet ensemble.

$$CAspect_i = \{A_0,...,A_j\} = \{\{Aspect_{00},...,Aspect_{0m}\}_0,...,\{Aspect_{j0},...,Aspect_{jv}\}_j\}_i \quad (4.1)$$

Nous écrirons l'opération de tissage T à un instant t de ces cascades comme dans l'ÉQUATION 4.19. Elle prend en entrée deux arguments : (1) l'application initiale Ass_0 et (2) un ensemble de n cascades d'aspects CA_n. Elle produit une application adaptée appelée Ass_{final}.

$$T_t(Ass_0, CA_n) = Ass_{final} \mid CA_n = \{CAspect_0,...,CAspect_n\} \quad (4.2)$$

En terme de variabilité, les possibilités offertes par une cascade peuvent être décrite par le diagramme de feature présenté en FIGURE 4.2. Tous les aspects d'une fonctionnalité sont autant de variantes de cette dernière et peuvent être combinés. Ils sont donc exprimés avec une relation de type « ou » et peuvent coexister. Par contre, les fonctionnalités qui composent la cascade sont reliées par une relation de type « et ».

A terme, comme pour les approches à aspects classiques, plus que la relation d'ordre de type précédence, des contraintes ou autres relations pourront être exprimées. Par exemple, il serait possible d'ajouter une relation permettant d'obtenir un équivalent à l'opérateur XOR des diagrammes de feature. Une autre possibilité serait d'utiliser des relations semblables à celles proposées dans les ConcurTaskTrees [155]. Les relations proposées ici dans les cascades d'aspects pourraient être exprimées comme un parallèlisme (imbrication) entre les aspects et une relation d'activation entre les fonctionnalités ; d'autres relations comme l'exclusion pourraient être exprimées. Cependant, si localement dans une cascade d'aspects il est possible de perdre la propriété de symétrie, puisque nous considèrons qu'une cascade est conçue en connaissance des aspects qui la compose, une contrainte doit être respectée : l'ensemble des entités d'adaptation et donc des cascades d'aspects doit être extensible, c'est-à-dire qu'il doit être possible d'ajouter une cascade d'aspects sans nécessairement tenir compte de celles qui le sont déjà.

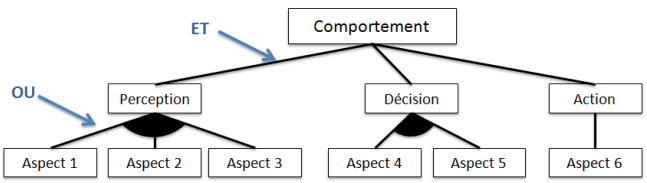


FIGURE 4.2 – Diagramme de features d'une cascade.

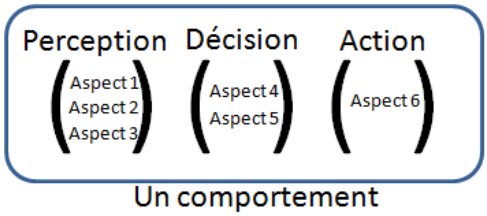


FIGURE 4.3 – Un comportement réflexe externe comme une cascade d'aspects.

Nous avons vu que l'adaptation compositionnelle du niveau réflexe externe a pour objectif de mettre en place des réflexes internes dans l'assemblage de base. Ces réflexes sont autant de comportements hybrides mêlant décomposition comportementale et fonctionnelle (FIGURE 4.3).

4.2.2.2 Combiner décomposition comportementale et fonctionnelle

Dans l'optique d'augmenter la réutilisabilité du code mis en place dans divers comportements, une décomposition hybride, entre décomposition comportementale et décomposition fonctionnelle, est réalisée. Il s'agit plus exactement de réaliser une décomposition fonctionnelle dans une décomposition comportementale. Un comportement se compose lui-même d'un ensemble de fonctionnalités (FIGURE 4.4). De cette manière, le code d'une fonctionnalité (voir une fonctionnalité complète) présente dans plusieurs comportements pourra être réutilisé. Dans le cadre de la prise en compte du contexte, comme en robotique, la décomposition fonctionnelle classique reposera donc sur trois grandes fonctionnalités : perception, décision et réaction.

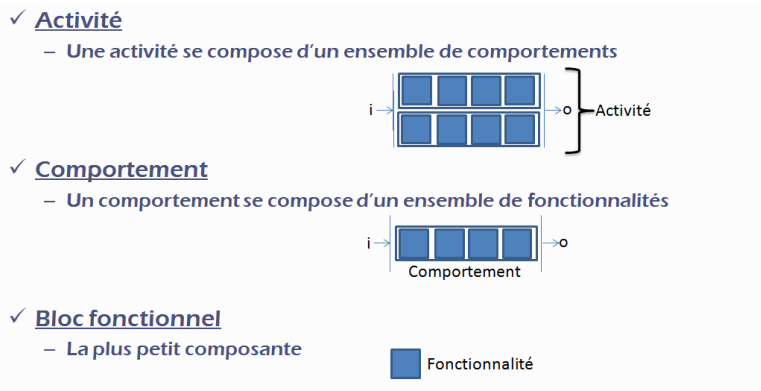


FIGURE 4.4 – Décomposition hybride.

L'approche hybride proposée tire partie des avantages des deux décompositions que sont en particulier : les capacités de maîtrise des dynamiques et d'indépendance offertes par la décomposition comportementale et la forte réutilisabilité offerte par la décomposition fonctionnelle. Plus particulièrement, l'approche hybride offre par rapport à l'approche comportementale :

- Une modularité dans les comportements et donc des facilités de maintenance mais aussi d'évolution d'un comportement,
- Des facilités de réutilisabilité de ces fonctionnalités et une meilleure séparation des préoccupations fonctionnelles d'un comportement.

Une cascade aura pour but d'instancier ces comportements réflexes internes dans l'application. Il s'agit donc d'une représentation abstraite d'un type de comportement hybride. Par conséquent, l'approche permet de concevoir le système avec une décomposition comportementale. Une telle décomposition facilite la définition de configurations du système pour des buts locaux (des comportements) qui mène plus facilement à la mise en place d'un système décentralisé. La décomposition fonctionnelle classique de ces comportements repose sur trois fonctionnalités principales : perception, décision et réaction. De fait, les cascades au niveau réflexe externe se baseront généralement sur trois ensembles pour ces trois fonctionnalités comme présenté en FIGURE 4.3. Les cascades déployées dans le tisseur pour les différents réflexes, pourront être combinées les unes avec les autres de manière à autoriser l'internalisation de réflexes comme des comportements hybrides.

Nous avons vu comment sont définies nos entités d'adaptation et de quelle manière elles permettent de gérer les préoccupations transverses hétérogènes et homogènes. Tout en autorisant de spé-

cifier le système à l'aide d'une décomposition comportementale et fonctionnelle. Une fois définies, ces cascades doivent être tissées d'une façon qui soit appropriée à la fois aux mécanismes de décision les plus flexibles comme les plus contraints. En particulier, les approches les plus flexibles nécessitent de la part du mécanisme de décision le respect des propriétés π_5 et π_6.

4.2.3 Composition opportuniste, déterministe et symétrique pour autoriser l'imprévisibilité

Dans les approches les plus flexibles, la variabilité ainsi que l'imprévisibilité de l'environnement requierent que l'ensemble des adaptations, et donc des cascades, soit **extensible** à l'exécution. Si l'adaptation paramétrée peut être étendue à travers l'utilisation de l'adaptation compositionnelle pour ajouter de nouveaux paramètres, l'adaptation compositionnelle nécessite de disposer de la possibilité d'ajouter à l'exécution de nouvelles adaptations. Dans les approches les plus flexibles, pour lesquelles la manière dont les adaptations sont composées est spécifiée le moins possible, il doit être possible de déployer de nouvelles adaptations, sans se préoccuper de celles qui le sont déjà. Ainsi, divers acteurs ont la possibilité de déployer à l'exécution leurs adaptations, décentralisant et distribuant les adaptations parmi les acteurs. Les adaptations doivent pour cela être **indépendantes** les unes des autres. Une adaptation ne doit donc pas : (1) empêcher une autre adaptation de s'appliquer, ni (2) déclencher explicitement le tissage d'une autre adaptation. Dans les approches les plus flexibles, l'imprévisibilité implique qu'il n'est pas possible d'anticiper et de connaître l'ordre dans lequel les adaptations vont être appliquées.

D'autre part, l'application des adaptations doit être réalisée de manière opportune, c'est-à-dire qu'une adaptation doit être appliquée lorsque l'infrastructure logicielle le permet et inversement doit être annulée lorsqu'une fonctionnalité nécessaire à son application est retirée. Ces adaptations doivent donc être combinées et tissées en fonction de l'application sur laquelle elles s'appliquent, respectant à minima une dynamique d'évolution : celle de l'infrastructure. Ainsi, un acteur ne peut connaître à priori l'ordre dans lequel ces adaptations vont être tissées. Il ne peut pas non plus nécessairement connaître a priori quelles seront les autres adaptations déployées. L'ordre dans lequel les adaptations sont appliquées, ne doit donc pas importer ni être anticipé. Cependant, le résultat de l'application d'un ensemble d'adaptations, quel qu'en soit l'ordre, doit être **déterministe** et **terminal**.

Pour toutes ces raisons, les entités d'adaptation doivent être déclenchées (appliquées/retirées) en réponse à une variation dans l'application sous-jacente, et donc combinées selon les **opportunités** offertes par cette dernière. Malgré ce mode de déclenchement et l'absence de contraintes entre les adaptations, pour que ces dernières soient indépendantes et donc que l'ordre dans lequel les elles sont appliquées n'importe pas, il est nécessaire que l'opération d'adaptation soit **déterministe** et **symétrique**.

Nous allons donc maintenant étudier de quelle manière s'opère le tissage des cascades d'aspects mais aussi comment les aspects d'une cascade peuvent être combinés de manière opportuniste, en fonction des évolutions de l'architecture de l'application sous-jacente. Nous verrons ensuite que cette approche permet de minimiser la combinatoire des adaptations à spécifier pour décrire un maximum de configurations. Pour que l'opération de tissage des cascades soit symétrique et déterministe, nous montrerons que l'opération de tissage des aspects qui compose chaque fonctionnalité doit l'être également. Nous identifierons alors quelles caractéristiques les aspects doivent respecter afin que leur opération de tissage soit symétrique et déterministe.

4.2.3.1 Tissage des cascades d'aspects

Nous avons vu précédemment qu'une cascade d'aspects se compose d'un ensemble de fonctionnalités, chacune d'entre elles se définissant comme un ensemble d'aspects. Le tissage d'une cascade d'aspects consiste à tisser successivement les fonctionnalités qui la compose. Le tissage d'une fonctionnalité, et donc d'un ensemble d'aspects sans contraintes entre eux, peut s'écrire comme le montre le formalisme (4.3). L'opération de tissage à un instant t est notée comme la fonction W_t. Elle prend comme arguments l'application Ass_j et un ensemble non-ordonné de n aspects A_n afin de produire une application Ass_{j+1}. L'indice j identifie que l'application est le résultat du tissage de la fonctionnalité destinée à être tissée en j^{me} position. Si $j = 0$ alors il s'agit de l'application initiale sans adaptation.

$$\forall t \; W_t(Ass_j, A_n) = Ass_{j+1} \mid A_n = Aspect_0, ..., Aspect_n \tag{4.3}$$

A la manière des cycles d'automates qui consistent en des phases successives d'acquisition de données, d'exécution et finalement de productions de sorties, nous qualifions l'opération de tissage d'une fonctionnalité de : *cycle de tissage*.

Définition 9 : Cycle de tissage
Un cycle de tissage est un processus qui a pour but d'adapter une application. Il prend en entrée une application et un ensemble d'aspects. Si au moins un des aspects passés en entrée peut s'appliquer alors il produit une application adaptée, sinon il ne modifie pas l'application initiale.

L'opération de tissage d'une cascade peut donc consister en un ou plusieurs cycles de tissages, en fait un cycle par fonctionnalité. L'ordre entre les fonctionnalités indique le cycle dans lequel elles seront tissées. Le cycle numéro 0 est toujours réalisé sur l'application initiale. Le cycle de tissage n est ensuite tissé sur l'application produite lors du cycle $n-1$. Le tissage en cascades d'aspects opère donc comme suit : les aspects destinés au premier cycle de tissage sont tissés sur l'application initiale. Puis, sur l'application résultante, sont tissés les aspects destinés au second cycle de tissage, et ainsi de suite jusqu'au dernier cycle de tissage (FIGURE 4.5).

Définition 10 : Application initiale
L'application initiale est un ensemble d'éléments applicatifs qui ne sont pas le fruit d'une adaptation.

Lors de chacun de ces cycles de tissages, les points de coupes de chaque aspect sont évalués afin de déterminer « si » et « où » l'aspect peut s'appliquer. Plus précisément, le processus de tissage d'une cascade peut donc être décrit comme suit. Dans un premier temps, le tisseur évalue les points de coupes de chacun des aspects destinés au premier cycle de tissage. Chacun d'entre eux pouvant être appliqué ou non indépendamment en fonction de l'application de base donnée en entrée au tisseur. Une fois toutes les modifications portées dans l'application, la nouvelle application adaptée sera passée en entrée au second cycle de tissage. Les points de coupe des aspects destinés à ce second cycle peuvent alors identifier des points de jonction dans l'application initiale mais aussi dans le résultat des modifications apportées lors du cycle précédent. Un point de jonction instancié par un aspect peut donc être utilisé par un autre aspect. De cette manière, les aspects peuvent être tissés en cascade, c'est-à-dire que l'application d'un aspect pour une fonctionnalité lors d'un cycle $n-1$ peut

être à l'origine du tissage d'un aspect du cycle n. Lors d'un nouveau déclenchement, l'ensemble du processus sera à nouveau réalisé, en recommençant par le cycle 0.

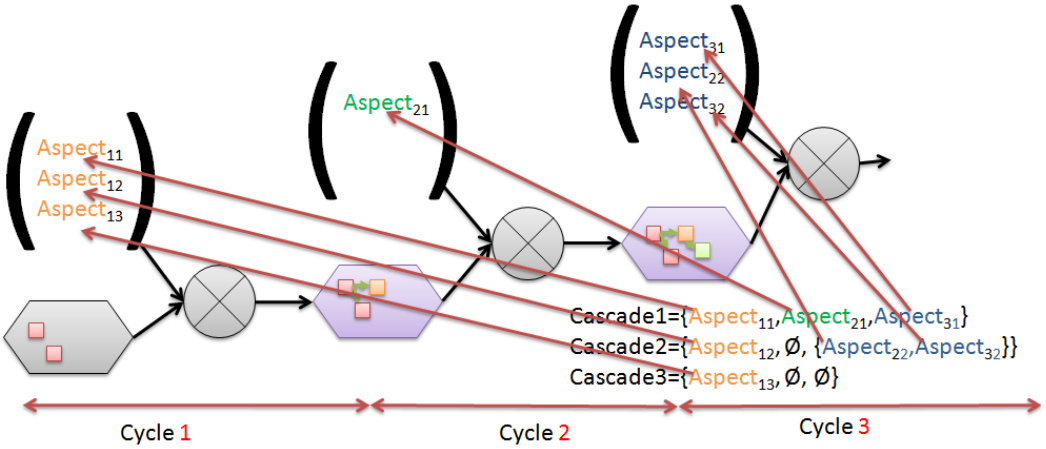


FIGURE 4.5 – Tissage de plusieurs cascades

Lorsque plusieurs cascades sont passées en entrée du mécanisme de tissage, l'opération de tissage consiste à faire l'union des fonctionnalités et donc des ensembles d'aspects destinés à un même cycle de tissage (FIGURE 4.5). Par exemple, si deux cascades sont données en entrée au tisseur et possèdent trois ensembles d'aspects pour les fonctionnalités de perception, décision et action, alors le premier cycle de tissage consistera à tisser sur l'application initiale l'union des deux ensembles pour la perception. Il en va de même pour les cycles suivants. A partir des équations précédemment définies 4.19 et 4.3, nous pouvons donc écrire l'opération de tissage de cascades comme dans l'EQUATION 4.17. Rappelons que T est l'opération de tissage des cascades d'aspects tandis que W_j représente l'opération de tissage des aspects pour une fonctionnalité destinées au cycle de tissage j. L'opération de tissage prend en entrée l'application initiale Ass_0 et l'ensemble des n cascades d'aspects CA_n. Elle consiste à réaliser j tissages successifs de telle manière que W_j prend en entrée le résultat de W_{j-1} ainsi que l'union, représenté par la fonction C, des n ensembles d'aspects A_{jn} destinés au cycle j.

$$
\begin{aligned}
&T(Ass_0, CA_n) = W_j(C(A_{j0},...,A_{jn}), W_{j-1}(C(A_{(j-1)0},...,A_{(j-1)n}),...,W_0(C(A_{00},...,A_{0n}), Ass_0))))\\
&C(A_{j0},...,A_{jn}) = A_{j0} \cup ... \cup A_{jn}\\
&C : E^j \rightarrow E
\end{aligned}
\tag{4.4}
$$

Afin de pouvoir ajouter facilement des cascades et que l'ordre dans lequel les cascades sont tissées ne soit pas important, il faut que leur opération de tissage soit déterministe et symétrique. Puisque le tissage de plusieurs cascades consiste, lors de chaque cycle de tissage, à faire l'union des aspects des différentes cascades qui sont destinées à ce cycle, l'opération de tissage des aspects dans un même cycle de tissage doit elle-même être symétrique.

4.2.3.2 Déterminisme et symétrie

Pour que le résultat du tissage de plusieurs aspects soit déterministe, sans exprimer de contraintes quant à leur composition, il est nécessaire que leur opération de tissage soit symétrique. La propriété

de symétrie se décompose elle-même en trois sous-propriétés : l'idempotence, l'associativité et la commutativité. L'idempotence signifie qu'une même instance d'une règle d'un aspect (par exemple, ajouter une liaison entre deux ports) ne peut être tissée qu'une fois. La commutativité permet de garantir que l'ordre dans lequel sont tissés les aspects n'a pas d'importance. Traditionnellement, le tissage des aspects n'est pas commutatif [16]. Enfin, l'associativité permet de regrouper des composition d'aspects sous la forme d'une même entité. Classiquement, l'opération de tissage des aspects est uniquement associative à droite [16].

Définition 11 : Propriété de symétrie du tissage d'aspects
La propriété de symétrie se décompose elle-même en trois sous-propriétés : idempotence, associativité et commutativité.
$Aspect_1 \otimes Aspect_2 = Aspect_2 \otimes Aspect_1$ (commutativité)
$(Aspect_1 \otimes Aspect_2) \otimes Aspect_3 = Aspect_1 \otimes (Aspect_2 \otimes Aspect_3)$ (associativité)
$Aspect_1 \otimes Aspect_1 = Aspect_1$ (idempotence)

Afin de satisfaire toutes ces propriétés, les aspects doivent respecter les contraintes suivantes :

1. Un point de coupe ne doit pas exprimer de règles négatives, c'est-à-dire qui requièrent l'absence d'un point de jonction. Ceci peut mener à la perte de la propriété de commutativité.
2. Un point de coupe ne peut identifier des éléments instanciés par d'autres aspects. Ceci peut entraîner la perte des propriétés de commutativité et d'associativité.
3. Un greffon ne peut supprimer un élément explicitement. Ceci peut être à l'origine de la perte des propriétés d'associativité et de commutativité.
4. La composition des règles se trouvant dans les greffons doit être symétrique.

Comme nous pouvons le voir, une des conséquences de la propriété de symétrie de l'opération de tissage est qu'un aspect ne peut réutiliser ou requérir l'absence d'un point de jonction instancié par un autre aspect. En effet, cela aurait pour conséquence d'introduire un ordre entre les aspects et entraînerait ainsi la perte de la propriété de commutativité de l'opération de tissage. Nous pouvons prouver ceci à l'aide du formalisme précédemment défini (ÉQUATION 4.3). Soit W l'opération de tissage des aspects pour une fonctionnalité, Ass une application et $aspect_1$, $aspect_2$ deux aspects. Le respect de cette propriété doit nous permettre d'écrire :

$$\begin{aligned} W(\{aspect_1, aspect_2\}, Ass) &= W(W(aspect_1, Ass), aspect_2) \\ &= W(W(aspect_2, Ass), aspect_1) \end{aligned} \tag{4.5}$$

Considérons maintenant que $aspect_1$ et $aspect_2$ sont deux aspects, supposons que :

1. L'application initiale est vide : $Ass_0 = \emptyset$
2. $Aspect_2$ ne peut être tissé que si l'élément c_2 est dans Ass, c'est-à-dire que les points de coupe de $aspect_2$ portent sur c_2

Posons également les hypothèses selon lesquelles $aspect_1$ ajoute un composant c_2 à Ass et $aspect_2$ ajoute un composant c_1 de telle manière que le tissage unique de $aspect_1$ modifie l'application tandis que le tissage unique de $aspect_2$ n'entraîne aucune modification de l'application :

$$\begin{aligned} &W(Ass, aspect_1) = Ass_1 \mid c_1 \in Ass_1 \\ &W(Ass, aspect_2) = Ass \end{aligned} \tag{4.6}$$

En considérant que $aspect_2$ peut réutiliser un élément instancié par $aspect_1$, l'EQUATION (4.5) n'est plus validé puisque le tissage de $aspect_2$ sur $aspect_1$ ne produit pas le même résultat que le tissage de $aspect_1$ sur $aspect_2$:

$$\begin{aligned} &W(W(aspect_1,Ass),aspect_2) = W(Ass_1,aspect_2) = Ass_2 \mid c_1,c_2 \in Ass_2 \\ &W(W(aspect_2,Ass),aspect_1) = W(Ass,aspect_1) = Ass_1 \mid c_1 \in Ass_1 \\ &W(W(aspect_2,Ass),aspect_1) \neq W(W(aspect_1,Ass),aspect_2) \end{aligned} \tag{4.7}$$

Le composant c_2 ne peut pas être réutilisé si nous souhaitons respecter la propriété de commutativité. Un aspect ne peut donc pas réutiliser un composant instancié par un autre aspect dans un même cycle de tissage. Le tissage de cascades multi-cycles permet de répondre à cette problématique. Si les aspects d'un même cycle de tissage n ne peuvent réutiliser des éléments qu'ils instancient, les aspects des cycles suivant le peuvent. Une preuve similaire peut être définie et nous permet de prouver que la suppression d'un élément de manière explicite par un greffon entraînerait la perte de la propriété de symétrie. Un élément doit donc être retiré lorsque l'aspect l'ayant instancié n'est plus appliqué.

Grâce à cette propriété de symétrie, l'opération de tissage peut être déterministe sans qu'il y ait d'ordre entre les aspects.

Définition 12 : Tissage déterministe

Pour un ensemble d'aspect et une application donnée, le résultat de l'opération de tissage déterministe sera toujours le même quelque soit l'ordre dans lequel ils sont tissés. L'opération de tissage de ces aspects peut être définie comme suit :

$$\forall t\ W_t(Ass_0,A_n) = Ass_{final} \mid Ass_{final}\ est\ unique \tag{4.8}$$

Tout comme l'opération de composition des aspects, la composition des cascades d'aspects doit également être symétrique afin qu'il soit possible d'ajouter de nouvelles cascades sans se préoccuper de l'ordre dans lequel elles vont être tissées. Prouvons par récurrence que l'opération de tissage des cascades est symétrique si l'opération de tissage des aspects l'est. Pour cela, prenons le tissage de deux cascades, d'après (4.17) nous pouvons écrire l'EQUATION 4.9. Rappelons que T désigne l'opération de tissage des cascades d'aspects CA_n (ici $n = 2$) et que W_j désigne l'opération de tissage des aspects destinés au cycle de tissage j. Rappelons également que C est la fonction réalisant l'union des ensembles d'aspects A_{jn} avec j l'identifiant du cycle de tissage auquel est destiné l'aspect et n l'identifiant de la cascade contenant cet ensemble $CAspect_n$.

$$\begin{aligned} T(Ass,CA_n) &= T(Ass,CAspect_1,CAspect_2) \\ &= W(C(A_{j1},A_{j2}),W(C(A_{(j-1)1},A_{(j-1)2}),...,W(C(A_{01},A_{02}),Ass))) \end{aligned} \tag{4.9}$$

Tout d'abord étudions le cas de base $j = 0$:

$$W(C(A_{01},A_{02}),Ass) = W(A_0,Ass) = Ass_0 \mid Ass_0\ est\ unique \tag{4.10}$$

Puisque l'opération de tissage W est symétrique tout comme C (puisque la fonction union est elle-même symétrique), alors la symétrie est vrai pour $j = 0$. Supposons par récurrence que la propriété

est vraie pour j et vérifions la pour $j+1$:

$$W(C(A_{(j+1)1},A_{(j+1)2}),Ass_j) = W(A_{j+1},Ass_j) = Ass_{j+1} \tag{4.11}$$

De la même manière, C et T étant des opérations symétriques, la symétrie est vraie pour $j+1$. L'opération de tissage des cascades est donc également symétrique lorsque l'opération de tissage des aspects est symétrique. Moins formellement, cela revient à dire qu'une cascade peut être considérée comme un aspect, puisque la fusion des aspects est symétrique, celle des cascades l'est également.

L'opération de tissage des cascades est donc déterministe (π_5) et symétrique si l'opération de tissage des aspects l'est également. L'ensemble des cascades est donc facilement extensible sans que l'ordre entre les cascades importe (π_6). Ainsi, en respectant ces propriétés logiques, divers acteurs peuvent aisément déployer de nouveaux comportements réflexes internes. Les cascades peuvent donc être appliquées dans n'importe quel ordre et, par conséquent, être composées de manière imprévisible. Ainsi, elles peuvent suivre les évolutions de l'infrastructure logicielle du système pour ne plus s'appliquer lorsqu'un dispositif nécessaire à leur application disparaît ou au contraire s'appliquer dès que possible.

4.2.3.3 Combinaisons opportunistes dirigées par l'infrastructure

Nous avons vu que le mécanisme d'adaptation associé aux comportements réflexes externes doit au moins respecter une dynamique : celle de l'infrastructure. Les aspects doivent donc être retirés dès que les éléments de l'infrastructure dont dépend leur tissage disparaissent. De la même manière, dans les approches par émergence, une adaptation, qui a été déployée parce que jugée pertinente pour la situation, doit être effectuée de manière opportune dès qu'elle peut l'être. De cette manière, les fonctionnalités décrites par une cascade peuvent être réalisées de manière opportune en fonction de l'infrastructure sous-jacente.

Ainsi, le premier levier pour déclencher le processus d'adaptation est l'apparition ou la disparition d'un élément de l'application et donc potentiellement du représentant d'un dispositif de l'infrastructure. Ce changement entraînera donc le processus de tissage en cascade des aspects. Le second levier permettant de déclencher le processus d'adaptation est la sélection ou désélection d'une cascade. On parle de ces phénomènes déclencheurs comme des perturbations.

Définition 13 : Perturbation
Une perturbation est un élément imprévu qui a pour effet de déclencher l'évaluation des cascades : apparition / disparition d'éléments dans l'application, déploiement / retrait de cascades.

Ces deux leviers sont présentés plus formellement en FIGURE 5.8. Dans cette figure, Δ_{ass} représente une perturbation dans l'application initiale, tandis que Δ_{CA} représente une perturbation dans l'ensemble des cascades.

Puisque lors de chaque cycle de tissage, les points de coupes de chaque aspect déployé sont évalués, le maximum d'aspects pour chaque fonctionnalité est appliqué en fonction de l'application initiale. Les aspects sont alors combinés de manière opportune. En ce sens, la perte dans l'infrastructure d'un dispositif et par conséquent de son représentant dans l'application, ne mène pas nécessairement à

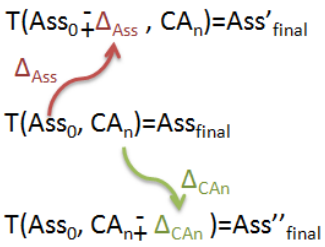


FIGURE 4.6 – Types de déclenchement d'un cycle de tissage.

la perte de la fonctionnalité dans l'application. Seules les parties de la fonctionnalités qui ne pouvaient pas être tissées ne le sont plus. De la même manière, il devient possible de fournir une alternative pour cette fonctionnalité, c'est-à-dire que si un dispositif est disponible et peut rendre un service équivalent à celui qui vient disparaître, il peut être utilisé à sa place. Grâce à cette approche, il n'est plus nécessaire d'écrire les transitions faisant passer d'une configuration du système à une autre et le nombre de règles d'adaptations à écrire est réduit.

4.2.3.4 Minimiser la combinatoire des règles d'adaptation tout en maximisant l'espace des configurations atteignables par le système

Puisque les aspects destinés à un même cycle de tissage sont exprimés avec une relation de type « ou » et peuvent coexister, lors d'un cycle, 2^n types de configurations peuvent être exprimés. Nous parlons ici de type de configuration car un aspect d'un certain type peut être instancié plusieurs fois dans l'application. L'espace des configurations ne peut donc pas être borné contrairement à celui des types de configurations. Les ensembles d'aspects composant une cascade étant ordonnés, et la relation entre ceux-ci étant de type « et », l'ensemble des types de configurations peut être décrit comme le produit de l'espace des configurations de chaque cycle comme cela est décrit dans la FIGURE 4.7.

$$C = \prod_{i=0}^{K} 2^{M(i)}$$

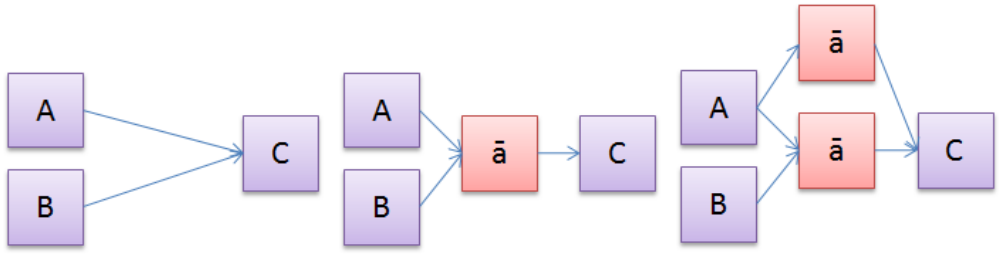
i : Identifiant du cycle de tissage
K : Nombre de cycles
M(i) : Nombre d'aspects devant être tissés lors du cycle numéro i
C : Nombre de combinaisons

FIGURE 4.7 – Nombre de combinaisons sans interactions.

Il est important de noter que lorsque les aspects sont tous indépendants les uns des autres, les combinatoires entre les approches reposant sur un ou plusieurs cycles de tissages sont équivalentes, à savoir 2^n configurations, n étant le nombre d'aspects. Par contre, nous avons vu qu'une des possibilités offerte par le tissage en cascade, lorsque qu'une cascade contient plusieurs ensembles d'aspects, est d'autoriser la réutilisation d'un point de jonction instancié lors d'un cycle de tissage par les aspects des cycles suivants. Ainsi, l'approche sur plusieurs cycles permet de décomposer en plusieurs aspects, dispersés sur plusieurs cycles, un aspect qui n'aurait pas pu l'être sur un cycle. En effet, la propriété de symétrie entraîne l'incapacité pour un aspect de réutiliser un élément produit par un autre aspect dans un même cycle de tissage, ce qui à pour conséquence de réduire les facultés de modularisation et par conséquent de réutilisabilité des aspects. Sur un cycle, un aspect ne peut

réutiliser une fonctionnalité instanciée par un autre aspect. De fait, il est possible de retrouver plusieurs fois le même code dans plusieurs aspects. La séparation des préoccupations mais aussi la réduction de l'entrelacement du code apporté par l'AOP en est d'autant limitée. Pour illustrer ceci, prenons un exemple portant sur des assemblages de composants présentés en FIGURE 4.8a. Nous pouvons écrire deux aspects sur un même cycle qui ont pour objectif d'ajouter des interactions entre des capteurs A et B et un actionneur C. Dans ce cas, nous pouvons écrire deux aspects. Maintenant, si nous rajoutons à l'exemple un composant intermédiaire dont le but est de faire la moyenne des valeurs fournies par A et B, l'assemblage souhaité est désormais celui décrit en FIGURE 4.8b. Sur un cycle, pour pouvoir lier A et B au composant de moyenne, il faut que l'ensemble de la configuration soit écrite dans un même aspect. Si nous souhaitons ajouter un nouveau capteur il faut que nous modifions ce même aspect. Pour écrire une nouvelle configuration, il est nécessaire d'écrire un autre aspect avec de nouveau la règle ajoutant la moyenne.

Sur plusieurs cycles il est possible d'écrire un aspect ajoutant le composant moyenne dans un premier cycle qui sera réutilisé par des aspects d'un cycle suivant ajoutant des liaisons entre ce composant et les actionneurs/capteurs. Ainsi, là ou il pouvait y avoir n aspects et 2^n configurations, il est possible d'exprimer grâce à cette décomposition $n+m$ aspects, m étant le nombre d'aspects ajoutés grâce à la décomposition. L'expression de ces contraintes entre aspects, comparativement à une approche sans contrainte, n'est donc pas nécessairement néfaste à la réduction de la combinatoire des règles d'adaptation à écrire ni pour la taille de l'espace des configurations décrit par ces adaptations. Cependant, ce n'est pas parce que nous pouvons écrire $n+m$ aspects que nous pouvons produire 2^{n+m} configurations.



(a) Cas (1) (b) Cas (2) (c) Cas (3)

FIGURE 4.8 – Illustration des limitations en terme de variabilité des AAs

En effet, avec cette décomposition, il s'agit d'ajouter une dépendance entre des aspects et de contraindre un peu plus les capacités d'adaptation. Dans ce cas, l'espace des solutions est réduit, puisque certains aspects ne sont tissés que lorsque d'autres le sont. Par exemple, si un aspect d'un cycle est réutilisé par tous les aspects des cycles suivants alors l'ensemble des types de configurations atteignable par le système peut alors être décrit comme dans la FIGURE 4.9.

Pour minimiser le nombre de règles à écrire tout en maximisant le nombre de configurations atteignable par le système, il faut donc organiser les fonctionnalités dans les cycles de tissage, en fonction de leur réutilisation. Si une fonctionnalité est plutôt une préoccupation transverse homogène alors elle est placée dans un cycle en amont dans le processus de tissage. Au dessus de ces dernières, nous tisserons les fonctionnalités variables qui sont des préoccupations transverses hétérogènes. Pour ne pas réécrire N variantes d'une même entité, nous plaçons donc une fonctionnalité après une

$$C = \prod_{i=0}^{K} 2^{M(i)-R(i)}$$

i : Identifiant du cycle de tissage
K : Nombre de cycles
$M(i)$: Nombre d'aspects devant être tissés lors du cycle numéro i
$R(i)$: Nombre d'aspects du cycle numéro i qui produisent un élément qui sera un requit par un autre aspect pour être tissés.
C : Nombre de combinaisons

FIGURE 4.9 – Nombre de combinaisons lorsqu'un aspect est réutilisé par tous ces successeurs.

autre si elle repose ou raffine la première. Une fonctionnalité repose nécessairement sur une autre si elle est appelée à être dupliquée et à varier tout en reposant sur un même élément. Dans l'exemple précédent, le composant moyenne ne dépend pas d'un dispositif et se trouve par conséquent peu sujet à variabilité. Par contre les capteurs et actionneurs peuvent varier fortement. Plutôt que d'écrire N variantes d'un même aspect avec la moyenne en fonction des capteurs et actionneurs présents, nous mettons d'abord en place la moyenne dans un aspect que nous ne réécrivons plus et nous décrivons seulement les variantes qui vont venir s'y rattacher dans des aspects qui se tisseront dessus.

Un autre point démontre que l'approche sur plusieurs cycles peut offrir des avantages en terme d'espace de configurations atteignables à partir d'un ensemble d'aspects. Sur un cycle, puisqu'il n'est pas possible pour un aspect de réutiliser un composant instancié par un autre aspect, nous avons vu que pour écrire certaines variantes du système, il est nécessaire de réécrire certaines règles, ici la règle ajoutant la moyenne. Ainsi lorsque nous souhaitons écrire les deux configurations de la FIGURE 4.8b il nous faut deux aspects. Mais, plus que d'écrire deux aspects, il faut également sélectionner uniquement l'aspect pertinent. En effet, puisque les aspects s'appliquent de manière opportuniste, c'est à dire dès qu'ils le peuvent, lorsque l'aspect décrivant la configuration la plus complète (i.e. plus gros point de coupe) peut être appliqué, tous les aspects dont les points de coupe $pointcut_i$ sont tels que $poincut_i \subset pointcut_j$ seront appliqués. Plus généralement, ceci peut s'écrire comme suit :

$$\forall\, aspect_1, aspect_2;\ \exists\, pointcut_i \in aspect_1,\ pointcut_j \in aspect_2 \mid poincut_i \subset pointcut_j \Rightarrow si\ aspect_2\ est\ tissé\ alors\ aspect_1\ l'est\ également \tag{4.12}$$

Ceci a pour conséquence que lorsqu'une variante d'une configuration (un aspect) d'une fonctionnalité du système est tissée, toutes ces sous-variantes (les aspects dont le point de coupe est une sous-partie de cet aspect) le sont également. Dans notre exemple, avec deux aspects dans un même cycle, nous obtenons la configuration présentée en FIGURE 4.8c

Même si les aspects sont symétriques, des interactions peuvent survenir entre eux et ce plus particulièrement au niveau des points de jonction partagés par plusieurs aspects. De plus, les cascades, en permettant à des aspects de réutiliser le résultat d'autres aspects, sont également à l'origine d'interactions entre aspects. Nous allons maintenant identifier ces interactions et de quelle manière elles peuvent être gérées.

4.2.3.5 Interactions entre cascades d'aspects

La propriété de symétrie des opérations de tissage d'aspects ou de cascades d'aspects limite le nombre des interférences qui peuvent survenir entre aspects ou cascades d'aspects. Entre les aspects destinés à un même cycle de tissage, nous avons vu qu'aucune contrainte n'est exprimée. Des interactions peuvent alors apparaître entre ces aspects. Dans [13], différents types d'interactions entre aspects sont définies, nous allons maintenant étudier, parmi celles-ci, lesquelles sont susceptibles de survenir. Grâce à la symétrie et au tissage opportuniste, les interférences de dépendance (un aspect requiert la présence d'un autre aspect pour fonctionner correctement) et d'exclusion mutuelle (deux aspects font la même chose, seul un aspect doit être mis en place) sont écartées. L'interférence de type renforcement qui intervient quand la présence d'un aspect renforce le bon comportement sémantique d'un autre aspect, dans le cadre de l'émergence, n'est pas problématique. Il reste des interférences de composition concurrente [5] qui surviennent lorsque plusieurs greffons interviennent sur le même point de jonction. Les conflits sémantiques, signifiant qu'un aspect requiert l'absence d'un autre aspect pour avoir un comportement sémantique correct, peuvent être laissées à la charge d'un mécanisme de décision associé aux cascades. Cependant, puisque ces interférences ne sont pas nécessairement gérées de manière explicites, le mécanisme permettant de les composer doit disposer d'un mécanisme de résolution automatique des conflits. Des travaux comme [64] ou [10] proposent de tels mécanismes que nous étudierons dans la mise en œuvre.

D'autre part, entre les aspects de différents cycles de tissages des interférences de dépendances peuvent apparaître. Un aspect, via l'ajout d'un élément, peut déclencher le tissage d'un autre aspect d'un cycle suivant de manière inappropriée et non-prévue. L'inverse n'est pas possible, puisqu'un aspect ne peut supprimer un élément qui était requis pour le tissage d'un autre aspect. Ces interférences surviennent lorsque les points de coupe d'un aspect sont trop permissifs ou obsolètes. Plus formellement, un aspect est dépendant d'un ou plusieurs autres aspects si l'opération de pointcut matching de cet aspect reconnait un élément instancié par un autre aspect Δ_n :

$$\begin{aligned} &W(CAspect_n, Ass_{n-1}) = Ass_n = Ass_{n-1} + \Delta_n \\ &\forall\, \Delta_n, \exists\, aspect_{n+1} \in CAspect_{n+1} \mid pointcutMatching(aspect_{n+1}, \Delta_n) \neq \emptyset \end{aligned} \tag{4.13}$$

Ce type d'interaction peut être géré à l'aide d'espaces de nommage. A chaque cascade peut être associé un nom et un espace de nommage. Par défaut, cet espace de nommage sera également celui des aspects qui les composent. Cependant, ils peuvent déclarer leur propre espace de nommage. Deux aspects avec le même nom de base, ne seront pas les mêmes s'ils n'appartiennent pas à la même cascade. Les éléments générés par un aspect appartiennent à son espace de nommage et ne peuvent être réutilisés sans connaissance de celui-ci. Ces dépendances peuvent donc être gérées avec 3 politiques :

1. Une cascade peut être considérée comme globale, c'est-à-dire que toute autre cascade pourra interagir avec elle. Dans ce cas, tous les autres aspects des autres cascades peuvent interagir avec elle.
2. Une cascade peut être dans un espace de nommage partagé par d'autres cascades. Dans ce cas, les aspects des autres cascades de cet espace de nommage pourront interagir avec les aspects qui la compose.
3. Une cascade peut être dans un espace de nommage qui n'est pas partagé, dans ce cas aucune interaction n'est autorisée avec les autres cascades.

L'approche proposée, malgré l'ensemble des propriétés logiques présentées, doit offrir des temps de réponse adaptés et maîtrisés.

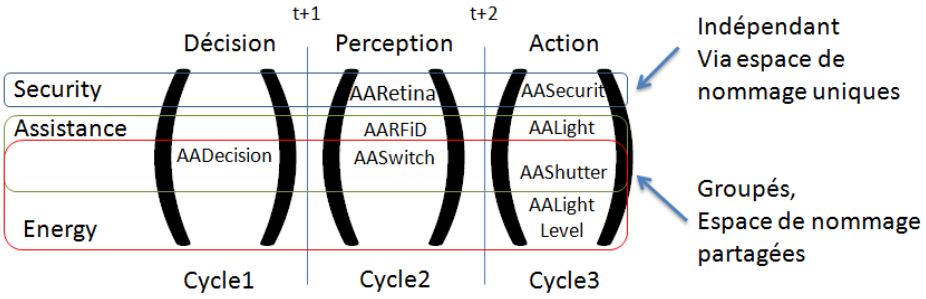


FIGURE 4.10 – Gestion des dépendances.

4.2.4 Temps de réponse maîtrisés et adaptés

Nous avons vu qu'une de propriétés du niveau réflexe externe de notre architecture sur quatre niveaux est d'autoriser des adaptations dans des temps de réponse finis et faibles. Le mécanisme d'adaptation doit donc offrir des temps de réponse maîtrisés et adaptés. Il doit alors être possible d'évaluer si les adaptations déployées ne sont pas trop importantes et par conséquent trop longues à mettre en place. Pour évaluer cela, différents critères peuvent être utilisés. Par exemple, dans le cadre des interactions hommes-machines, il est considéré que la « latence utilisateur » est dans le pire des cas de 100ms. Bérard et Crowley [138] proposent alors que la latence des systèmes hautement interactifs doit être moitié moins importante, c'est-à-dire de l'ordre de 50ms. Dans le domaine de la domotique, une latence d'une seconde est tolérée. Le mécanisme d'adaptation doit donc offrir des temps de réponse finis qui doivent pouvoir être comparés à ces bornes. Nous évaluerons en terme de temps de réponse notre mise en œuvre des cascades.

4.2.5 Synthèse

L'approche que nous proposons nous permet de répondre aux différentes contraintes que nous nous étions fixées. Les cascades en reposant sur des aspects dont l'opération de tissage est symétrique offrent les avantages suivants :

Indépendance Grâce à la symétrie de l'opération de tissage, les adaptations sont indépendants les unes des autres mais aussi auto-suffisantes. Une cascade ne peut empêcher une autre de s'appliquer ni ne peut déclencher explicitement le tissage d'une autre cascade. Il en va de même entre les aspects qui composent les cascades. Les interférences possibles entre cascades interviennent soit au niveau des points de jonction partagés et doivent être gérées par le mécanisme de composition ; soit par dépendance entre cascades via les points de coupes et peuvent être gérées à l'aide de la notion d'espace de nommage.

Extensibilité au runtime Un système, peut donc déployer des cascades sans se préoccuper de celles qui le sont déjà. De cette manière, divers acteurs ont la possibilité de déployer au runtime leurs cascades dans un même tisseur distribuant et décentralisant ainsi les règles d'adaptation parmi les acteurs. Règles qui seront centralisées localement dans un tisseur lors de leur déploiement. Dans l'architecture à quatre niveaux, un des objectifs des différents mécanismes de décision utilisés sera de garantir la cohérence sémantique des adaptations déployées. En fonction des temps de réponse nécessaires pour cela, le mécanisme pourra être déployé dans le ou les niveaux adaptés.

Composition non-anticipé Un designer ne peut connaître à priori l'ordre dans lequel ces adaptations vont être tissées. En effet, ce dernier ne peut connaître l'ordre dans lequel vont apparaître/disparaître des composants dans l'assemblage. D'autre part un acteur ne peut connaître *a priori* les autres adaptations déployées. La manière dont les entités d'adaptations doivent être composées ne peut donc être anticipée. Les cascades et aspects permettent, grâce au respect de la propriété de symétrie de l'opération de tissage d'avoir une composition non-anticipée.

Opportunisme L'ensemble des propriétés précédemment évoquées permettent de tisser (détisser) une cascade et les aspects qui la compose dès que l'infrastructure d'une application le permet (ne le permet plus) de manière indépendante pour chaque aspect.

Multi-dynamicité Ceci permet d'avoir une approche de la prise ne compte du contexte multi-dynamiques garantissant a minima le respect d'une dynamique : celle de l'infrastructure. Une fois ce pré-requis assuré, divers mécanismes logiques poursuivant leurs propres dynamiques pour leurs préoccupations et reposant sur les aspects peuvent être mis en place [75]. Chacun de ces mécanismes peut alors se concentrer et ne considérer que ce qui est pertinent pour lui.

Nous avons donc vu que pour réaliser des cascades telles que nous les décrivons il faut que les aspects composant la cascade respectent un ensemble de propriétés logiques. Les travaux précédents et en cours de l'équipe sur les Aspects d'Assemblages [114] définissent des aspects permettant de réaliser des adaptations dynamiques d'assemblages de composants de manière non-invasive. Ils proposent une opération de tissage symétrique qui peut être déclenchée en fonction des variations intervenant dans l'assemblage de composants sur lesquels ils portent. Nous allons donc voir maintenant comment utiliser ces Aspects d'Assemblages (AAs) pour réaliser des cascades d'AAs. Nous étudierons en détail les processus de tissage afin de vérifier leur respect des propriétés logiques et temporelles requises. Nous présentons également les différents modèles et transformations utilisées à l'exécution par les cascades d'AAs.

4.3 Les Aspects d'Assemblage

Les Aspects d'Assemblage sont issus des travaux précédents et en cours de l'équipe [64, 114]. Les contributions apportées à ces travaux dans le cadre de cette thèse sont un formalisme, une modélisation ainsi qu'une description algorithmique et temporelle de l'approche. Les Aspects d'Assemblages sont un mécanisme d'adaptation inspiré de l'AOP qui permet de réaliser des adaptations compositionnelles d'assemblage de composants tout en garantissant des propriétés logiques de symétrie et de déterminisme, ainsi que des propriété temporelles.

4.3.1 Principes et formalisation

Les Aspects d'Assemblage (AAs) sont un modèle inspiré de la Programmation Orientée Aspects (AOP). Si l'AOP permet de modifier un programme, les Aspects d'Assemblages permettent de modifier la structure d'assemblages de composants. Il s'agit de tisser, dynamiquement (π_1), en réaction à des variations dans l'assemblage de composants, des schémas d'adaptation en suivant le principe de modifications structurelles. Le tisseur inclut également un mécanisme permettant de fusionner ces schémas d'adaptation selon une logique bien définie lorsqu'ils sont en interaction les uns avec les autres. Afin de ne pas répercuter directement les modifications de chaque aspect dans l'application puis de re-modifier l'application pour en résoudre les interactions, le tisseur travaille sur une représentation abstraite, un modèle, d'assemblage de composants (π_3). Nous présenterons plus en détails

ces mécanismes dans les sections suivantes. Afin d'en simplifier la lecture, dans la suite de cette section, nous parlerons d'assemblage de composants plutôt que de modèle d'assemblage de composant. Le mécanisme de résolution des interférences entre AAs, permet de les tisser sur des applications qui ne sont pas forcément connues lors de leur écriture. S'ils violent la propriété d'encapsulation des composites qu'ils adaptent, ils ne violent pas la propriété de boite noire des composants sur étagère utilisés dans ces composites. Les modifications structurelles peuvent donc consister à ajouter des composants ou des liaisons entre ces composants (π_2). Les points de jonction des AAs sont toutes les entités d'un assemblage qui représentent structurellement une application : les ports des composants. Comme les aspects classiques, les AAs se composent d'une section point de coupe et d'un greffon. Un greffon décrit des modifications structurelles à apporter à un assemblage de composants tandis que les points de coupe permettent d'identifier les points de jonction sur lesquels porteront ces modifications. Un aspect d'assemblage peut donc être défini plus formellement comme suit (l'indice i identifie l'AA) :

$$AA_i = (pointcut_i, advice_i) \tag{4.14}$$

4.3.1.1 Les points de coupes

Les points de coupes sont définis comme des ensembles de filtres portant sur les méta-données associées aux points de jonctions (nom, type ...). Chaque filtre construit une liste de points de jonction le vérifiant. Ils permettent de rechercher et sélectionner dans l'assemblage les points de jonction sur lesquels vont porter les modifications décrites dans les advices. Cette liste est alors associée à une variable qui aura pour but de faire la liaison entre l'assemblage de composants représentant l'application et une configuration abstraite décrite dans son greffon. Un point de coupe ne peut exprimer de règle négative, c'est-à-dire qui requière l'absence d'un composant (cf. SECTION 4.2.3.2). Les éléments composants toutes ces listes peuvent être associés selon diverses combinaisons de telle manière qu'une combinaison contient un points de jonction de chaque filtre. Le greffon de l'aspect sera alors dupliqué autant de fois qu'il y a de combinaisons.

Définition 14 : Duplication d'AA
Un Aspect d'Assemblage est dupliqué lorsqu'il peut être appliqué en plusieurs endroits de l'assemblage. Un AA peut être dupliqué lorsque plusieurs points de jonction vérifient une même règle de point de coupe. Son greffon est alors instancié plusieurs fois.

Un même AA peut donc adapter en plusieurs endroits une même application. Les points de coupe sont le moyen de décrire plusieurs reconfigurations d'une application avec un haut niveau d'abstraction. Cette abstraction permet d'appliquer les reconfigurations sur des assemblages de composants qui ne sont pas connus a priori. Un point de coupe i de l'AA d'indice i est défini plus formellement dans l'EQUATION 4.15 comme un ensemble de règles caractérisées par deux indices : (1) i qui identifie l'AA auquel elle appartient et (2) j l'identifiant de la règle.

$$poincut_i = \{Rule_{i,0}, ..., Rule_{i,j}\} \tag{4.15}$$

4.3.1.2 Les greffons

Les greffons des Aspects d'Assemblage ne décrivent pas des morceaux de codes qui doivent être injectés dans le code d'une application. Ils définissent des ensembles de composants et de liaisons

qui doivent être tissées dans un assemblage de composants. Ces reconfigurations sont basées sur les points de jonction obtenus à l'aide des points de coupe. Pour ce faire, ces greffons sont composés d'un ensemble de règles abstraites basées sur les variables des points de coupe. Ces règles définissent quels composants et liaisons doivent être instanciés. Lorsque les variables sont substituées par des points de jonction, le tisseur d'aspects d'assemblage produit des instances de greffon. L'ensemble des instances de greffons des différents AAs déployés dans le tisseur sont ensuite composées les unes avec les autres. Il est important de noter qu'un greffon ne peut contenir une règle ayant pour objectif de supprimer un composant (cf. SECTION 4.2.3.2). Les composants instanciés par une instance de greffon sont retirés lorsque celle-ci est détissée. Les autres composants ne peuvent être retirés et représentent l'infrastructure sur laquelle les AAs peuvent s'appliquer. Un AA qui retirerait un tel composant pourrait empêcher le tissage d'autres AAs. Nous verrons plus loin comment grâce à l'existence de composants de sémantique connue, un designer peut gérer la manière dont vont être composées des instances de greffon grâce au mécanisme de fusion inclus dans le tisseur. Un greffon $advice_i$ appartenant à l'AA i, peut être défini plus formellement comme un ensemble de règles caractérisées par deux indices : (1) i qui identifie l'AA auquel elle appartient et (2) w l'identifiant de la règle.

$$advice_i = \{ARule_{i,0}, ..., ARule_{i,w}\} \tag{4.16}$$

4.3.1.3 Méta-modèle d'Aspect d'Assemblage

Les Aspects d'Assemblage sont également conformes à un méta-modèle qui permet de ne pas les rendre dépendant d'un langage en particulier. De cette manière, divers langages peuvent être utilisés dès lors qu'ils peuvent être transformés dans un modèle conforme au méta-modèle des AAs décrit en FIGURE 4.11. De cette façon, une préoccupation peut être écrite à l'aide du langage le plus adapté. Un exemple de langage utilisé pour écrire les Aspects d'Assemblage est ISL4WComp proposé par Daniel Cheung dans sa thèse [64]. Dans le cadre du projet Faros [130], j'ai pu réaliser ces modèles et méta-modèles d'AA et écrire à l'aide de l'outil Kermeta [128] la transformation : *modèle d'Aspect d'Assemblage* $\longrightarrow$ *ISL4WComp*. Dans ce méta-modèle, un AA possède un nom et appartient à un espace de nommage. Plusieurs AAs peuvent donc appartenir à un même espace de nommage. Généralement on associera à un fournisseur d'AAs un ou plusieurs espaces de nommage.

Définition 15 : Identité d'un AA
Un AA est identifié par son nom et son espace de nommage. Autrement dit, si deux AAs ont un nom et un espace de nommage identiques alors il s'agit du même AA.

Nous voyons dans ce méta-modèle que les points de coupe sont composés de filtres associés à des variables et les greffons sont des ensembles de règles. Nous retrouvons trois types de règles : (1) les règles pour l'instanciation de composants boites noires, aussi appelés composants locaux, (2) les règles de création et (3) de réécriture d'interactions entre composants. Grâce à l'utilisation d'opérateurs dont la sémantique est connue, il est possible de spécifier de quelle manière plusieurs règles de type (2) ou (3) peuvent être composées. Ces opérateurs prendront la forme de composants dans l'assemblage résultant du tissage. Nous verrons en SECTION 4.3.4 comment, grâce à l'utilisation de ces opérateurs, l'opération de composition de règles d'adaptations peut être symétrique (π_6). L'ensemble de ces opérateurs n'est pas figé.

Définition 16 : Règles d'instanciation de composants

Ces règles peuvent ajouter un composant sur étagère vu comme une boite noire. Elles ne permettent pas de supprimer des composants, cela aurait pour effet d'autoriser des interactions entre greffon d'un AA et points de coupe d'autres AAs. Selon l'ordre dans lequel les AAs seraient tissés le résultat ne serait plus le même. Un composant peut être retiré lorsque l'AA qui l'a instancié est détissé.

Définition 17 : Règles de création d'interactions entre composants

Ces règles ont pour membre gauche un port de sortie et aboutissent dans leur membre droit à des ports d'entrée. Elles peuvent utiliser des opérateurs du langage mais ne permettent pas explicitement de supprimer une interaction entre composants. Par contre, leur composition peut avoir pour effet de remplacer un liaison par un sous-assemblage semblable à un connecteur complexe voir de supprimer une liaison. Une liaison instanciée par un AA est retirée lorsque celui-ci est détissé.

Définition 18 : Règles de réécriture d'interactions entre composants

Ces règles ont pour membre droit et membre gauche des ports d'entrée. Elles peuvent utiliser des opérateurs de sémantique connue. Elles signifient que l'on souhaite réécrire ou remplacer par des sous-assemblages les appels au port membre gauche de la règle.

Nous voyons donc que des composants ne peuvent être retirés que si les AAs à l'origine de leur instanciation sont détissés. Les interactions peuvent également être retirées par ce biais mais aussi lors de l'opération de composition de plusieurs règles ou encore de manière générique par réécriture. En aucun cas des points de jonctions et liaisons ne peuvent être explicitement identifiées pour suppression.

Les AAs permettent donc de décrire des variantes de l'architecture d'un système, mais des variantes à un niveau « type », c'est-à-dire que ces variantes peuvent être instanciées de diverses manière puis tissées dans l'application pour en modifier la structure. Différents langages peuvent donc être utilisés afin d'exprimer des greffons d'aspects d'assemblage. Nous avons vu qu'ISL4WComp est un de ces langages.

4.3.1.4 ISL4WComp

Le langage de greffons ISL4WComp s'inspire du langage de spécification d'interactions ISL (Interaction Specification Language) qui décrit des schémas d'interactions indépendants entre des objets [134]. ISL4WComp adapte ces spécifications pour la prise en compte d'interactions basées sur des messages ou des événements, entre des composants dans les Aspects d'Assemblage. Ce langage a la particularité d'être composable : plusieurs instances de greffons peuvent se composer et fusionner en un seul assemblage combinant leur comportement respectif. Les règles ISL4WComp sont identifiées par deux mots clés, le ' :' pour la création d'instance et le '->' pour la réécriture et la création de liaison. Les parties droites des règles de modification de liaison sont sujettes à variation. La syntaxe générale de ces règles est présentée dans le tableau 4.2.

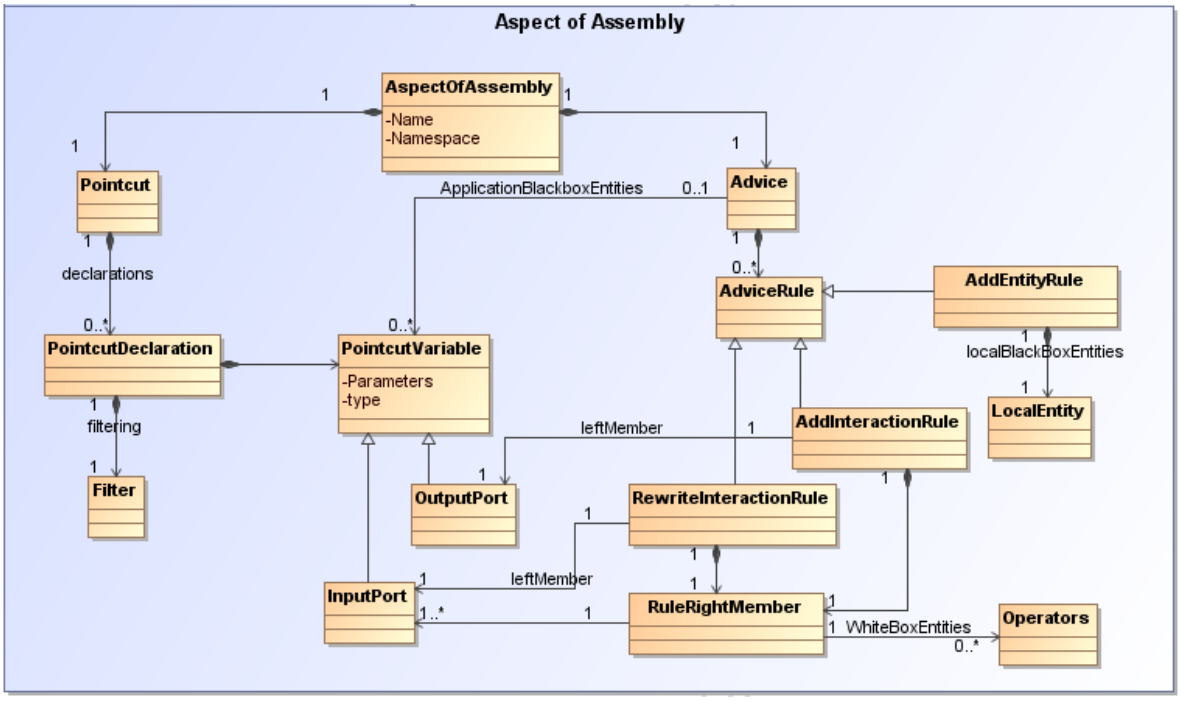


FIGURE 4.11 – Méta-modèle des Aspects d'Assemblage

Certains opérateurs du langage comme *call* et *delegate* permettent de contrôler la manière dont la composition des schémas d'assemblage va être effectuée. Ces mots-clés, associés aux opérateurs de *séquence* ou de *parallélisme*, se rapprochent alors des mots-clés comme *before*, *around* et *after* des points de coupe de l'AOP classique. Cependant, ils peuvent tous être fusionnés dynamiquement. Nous présenterons par la suite, le mécanisme de fusion associé à ce langage et comment les opérateurs du langage peuvent être fusionnés s'ils sont en conflit et ce de manière symétrique. Le Tableau 4.3 présente les mots clés et opérateurs d'ISL4WComp.

La FIGURE 5.13 présente un exemple de greffon écrit avec ISL4WComp. Dans cet exemple et les suivants, afin de clairement différencier les points de coupe et les greffons, ces sections sont respectivement séparées par les titres *pointcut* et *advice* . Il définit une adaptation indépendante pour une application de domotique, liant un interrupteur de n'importe quel type à une lampe. Les deux dispositifs sont représentés par des composants dans l'assemblage existant, et fournissent des fonctionnalités par leurs ports.

```
Pointcut:
    sender:=/switch[[:digit:]].^on/
    receiver:=/light[[:digit:]].SetStatus/
Advice:
    schema switch_light (sender, receiver) :
        sender -> (
            receiver.SetStatus)
```

FIGURE 4.12 – Exemple d'AA écrit avec ISL4Wcomp.

TABLE 4.2 – Syntaxe des règles ISL4WComp

$comp : type$	Règle d'instanciation de composants boite noire
$comp : type\ (prop = val, \ldots)$	Règle d'instanciation de composants boite noire avec initialisation de propriétés du composant créé.
port_requis → (port_fourni)	Règles de création d'interactions entre composants. Le mot clé → est le séparateur de partie gauche et droite. La partie droite peut contenir des opérateurs du langage et ainsi décrire un assemblage de composante.
port_fourni → (port_fourni)	Règles de réécriture d'interactions entre composants. Le mot clé → est le séparateur des parties gauche et droite. La partie droite peut contenir des opérateurs du langage et ainsi décrire un assemblage de composante.

Le greffon ci-dessous (FIGURE 4.13) propose d'adapter ce fonctionnement en prenant en compte un capteur de luminosité en n'allumant la lampe que lorsque la luminosité est trop faible, pour éviter une surconsommation énergétique par exemple.

Nous allons maintenant étudier le code de cet advice. Le greffon est appelé *brightness_light*. Les deux identifiants *condition* et *sender*, entre parenthèses à la première ligne, représentent les composants variables fournis par le calcul des points de coupe. Elles seront remplacées par les points de jonction identifiés lors du pointcut matching.

Ce greffon illustre les trois types de règles qui peuvent être mis en jeu dans un AA. Aux lignes 7, et 8, des composants boites noires sont ajoutés par l'AA. Pour le composant seuil (ici appelé *threshold*), une propriété est initialisée explicitement. Une propriété est une variable publique d'un composant, accessible par son interface. Les lignes 10 à 13 définissent une règle de création d'interaction. Tous les événements envoyés par le port de sortie *on* seront réécris. Cette règle utilise l'opérateur *if*, cela signifie qu'un composant de type *if* sera instancié par le tisseur. La condition, qui doit être évaluée par ce composant, est obtenue par l'appel à la méthode *IsReached* du composant boite noire *threshold*. Si la condition est vraie, rien n'est fait, par contre, si elle est fausse alors le lien réécrit est réalisé. Dans notre cas, il s'agit de l'appel à la méthode SetStatus présentée dans l'AA précédent (FIGURE 5.13). La règle définie à la ligne 14 permet de définir de nouvelle interactions. Elle relie la sortie *Brightness_Value_Evented_NewValue* du composant *brightness* à l'entrée de la méthode *set_Value* du composant boite noire *threshold* ainsi qu'à la méthode *set_Text* du composant boite noire *t2*. Ainsi, la valeur du seuil utilisée par le composant *if* varie en fonction de la luminosité.

Nous allons maintenant voir plus en détail le mécanisme de tissage et de quelle manière il permet de prendre en compte dynamiquement les apparitions/disparitions de composants dans un assemblage. Par conséquent, lorsqu'appliqué sur des middleware orientés service pour lesquels les apparitions/disparitions de dispositifs sont mappées directement en apparition/disparition de composants, il permet de prendre en compte les apparitions/disparitions de services dans l'infrastructure logicielle d'un assemblage. Ce processus est réalisé tout en garantissant la construction d'une application **cohérente** avec des **temps de réponse maîtrisés et raisonnables de manière non-anticipée**.

TABLE 4.3 – Mots clés et opérateurs d'ISL4WComp

	Mots clés / Operateurs	Description
Types de ports	*comp.port*	Le '.' sépare le nom d'une instance de composant du nom d'un port. Cette notation désigne un port fourni (méthode).
	comp.^ port	Le '^ ' en début de nom de port désigne un port requis (événement).
Opérateurs (propriétés de symétrie, résolution de conflits)	... ; ...	Exprime une séquence.
	... || ...	Exprime un ordre indifférent (parallélisme).
	`if (condition) {...} else {...}`	Exprime une *condition* évaluée par un composant boite noire.
	`nop`	Ne fait rien (utile pour un if dont une des deux actions n'est pas utilisée).
	`call`	Permet de réutiliser la partie gauche dans la règle de réécriture de liaison.
	`delegate`	Permet de spécifier une unicité sur une interaction lors des conflits.

```
Pointcut:
    sender:=/switch[[:digit:]].^on/
    receiver:=/light[[:digit:]].SetStatus/
    condition:=/ *.^Brightness_Value_Evented_NewValue/
Advice:
    schema brightness_light (sender, condition) :
        threshold: 'BasicBeans.Threshold' ( threshold = 10 )
        t2: 'System.Windows.Forms.TextBox'

        sender -> (if (threshold.IsReached)
                        {nop}
                   else
                        {call})
        condition -> (threshold.set_Value ; t2.set_Text)
```

FIGURE 4.13 – Exemple de greffon écrit avec ISL4Wcomp utilisant des opérateurs du langage.

4.3.1.5 Cycle de tissage

Un cycle de tissage d'Aspects d'Assemblage prend en entrée un assemblage de composants, que nous appellerons assemblage initial, et un ensemble d'Aspects d'Assemblage. Il produit un assemblage final, un assemblage adapté. Les cycles de tissages sont indépendants les uns des autres. Chaque cycle de tissage est réalisé sur un assemblage initial vierge de toute adaptation.

Définition 19 : Assemblage initial
L'assemblage initial est un ensemble de composants et de liaisons entre ces composants qui ne sont pas le fruit d'une adaptation.

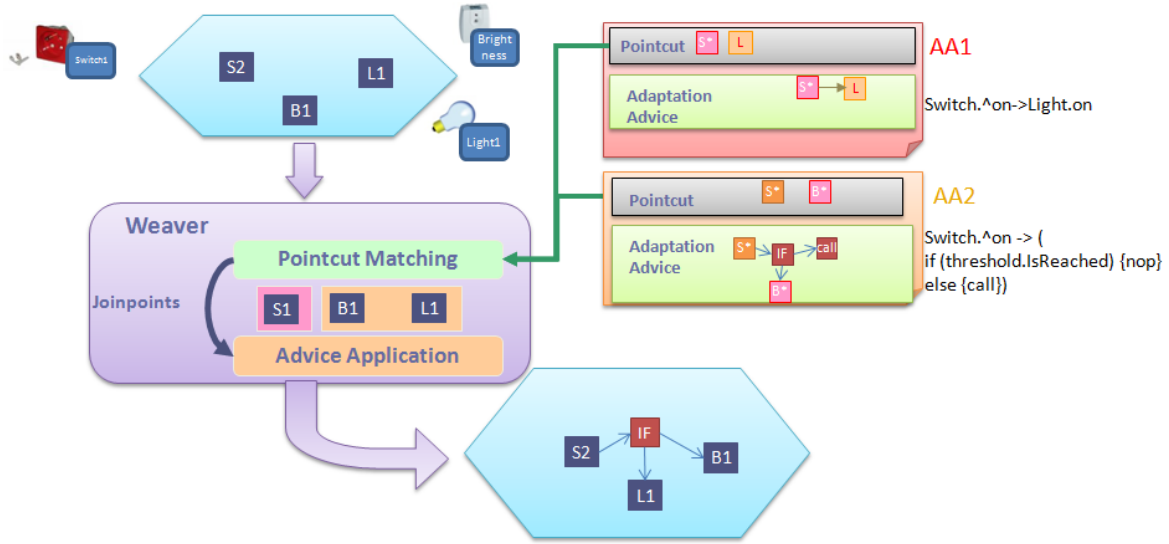


FIGURE 4.14 – Processus de tissage d'Aspects d'assemblage.

Durant un cycle, le processus de tissage peut être découpé en trois étapes principales. Dans un premier temps le tissage réalise l'étape dite de pointcut matching. Il s'agit d'une fonction qui prend en entrée un ensemble de composants provenant de l'assemblage de base et les points de coupe des AAs sélectionnés. Le but de cette fonction est de trouver les points de jonction sur lesquels les greffons pourront être tissés. La seconde étape, appelée la fabrique de greffon, consiste à générer des instances de greffon en remplaçant les composants abstraits des greffons par les points de jonctions obtenus grâce au pointcut matching. Les instances de greffon obtenues décrivent les reconfigurations qui devront être apportées à l'application de base. Enfin, le moteur de composition, fusionne toutes les instances de greffon obtenues et celles représentant l'assemblage de base dans l'optique de produire le nouvel assemblage adapté. Ce processus de tissage peut être écrit en respectant le formalisme précédemment défini (4.3) comme suit :

$$W_t(Ass_0, A_n) = Ass_{final} \mid A_n = \{AA_0, ..., AA_n\} \tag{4.17}$$

Un cycle de tissage peut être déclenché de deux manières (FIGURE 5.8) :

Dirigé par l'utilisateur Par une sélection/désélection des adaptations décrites par des AAs données en entrée à l'exécution au tisseur. Si un AA était appliqué, son retrait entraîne un nouveau cycle de tissage. Si un nouvel AA est sélectionné cela entraîne également un nouveau cycle de tissage.

Dirigé par le contexte La seconde manière de déclencher un cycle de tissage est l'ajout/retrait de composants dans l'assemblage initial donné en entrée au tisseur. Quand un nouveau composant apparaît ou disparaît, par exemple lorsque de nouveaux dispositifs entrent dans l'infrastructure de l'application, un nouveau cycle de tissage est lancé et seuls les AAs qui peuvent être appliqués sur ce nouvel assemblage seront tissés.

Un point essentiel pour la réactivité de ce mécanisme est qu'il ne requiert pas d'informations relatives à l'infrastructure logicielle, mais avec une dynamique qui lui est propre l'infrastructure impose son rythme via ce type de déclenchement. Cependant durant un cycle de tissage le système ne tolère plus de nouvelles perturbations, celui-ci termine son cycle avant de prendre en compte les nouvelles évolutions.

Les AAs peuvent donc se trouver dans différents états (Fig. 4.15). Les deux types de déclenchements ont pour effet de changer l'état d'un AA, chaque déclenchement menant à un état différent. À l'origine un AA est dans un état désélectionné, c'est-à-dire que l'utilisateur ne souhaite pas le voir appliqué. Dans un tel cas, ces points de coupe ne sont pas évalués. Lorsqu'un AA est sélectionné alors ces points de coupe sont évalués et le seront à nouveau pour chaque modification de l'assemblage sur lequel va porter l'AA. Si des points de jonctions vérifient les points de coupes alors l'AA passe en état applicable avant d'être tissé. Les AAs qui n'étaient pas applicables peuvent le devenir si une nouvelle entité apparaît dans l'assemblage de l'application. De même, ceux qui étaient tissés peuvent redevenir inapplicables, mais restent sélectionnés, si une entité identifiée dans leurs points de coupe venait à disparaître.

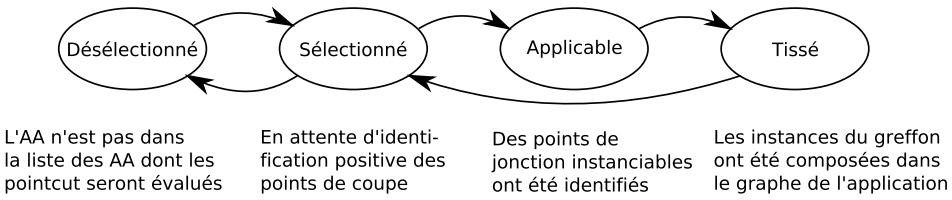


FIGURE 4.15 – Cycle de vie d'un Aspect d'Assemblage [74].

Ceci nous permet d'avoir une approche de déclenchement multi-dynamiques. En effet divers mécanismes, poursuivant leur propre dynamique, peuvent sélectionner ou déselectionner des AAs. C'est à dire spécifier les adaptations souhaitées. Mais les adaptations sont toujours réalisées en respectant à minima une dynamique : celle de l'infrastructure de l'application. En effet celle-ci est prioritaire, une application ne doit pas, par exemple, utiliser des services qui ne sont plus dans son infrastructure. Les processus de pointcut matching et de synchronisation du modèle d'une application avec l'application s'exécutant permettent de toujours adapter en fonction des possibilités offertes par l'infrastructure sous-jacente. Ainsi, un aspect sélectionné doit toujours être identifié comme dans un état « applicable », avant d'être tissé.

4.3.2 Le tisseur d'Aspects d'Assemblage

Le tisseur d'Aspects d'Assemblage est l'entité logicielle qui réalise l'ensemble de ces traitements. Nous allons maintenant l'étudier plus en détails ainsi que le formalisme introduit dans cette thèse. L'architecture du tisseur peut être décomposée en deux parties. La première traite des AAs et a pour but de transformer les greffons de ces derniers en assemblages de composants. Pour ce faire, le traitement sur les AAs repose sur trois étapes. Le pointcut matching ① qui identifie l'ensemble des points de jonction sur lesquels peut s'appliquer l'AA. Les greffons des AAs utilisent ensuite ces points de jonction de manière à ce que chaque variable soit remplacée par un point de jonction. Ceci est réalisé par la fabrique de greffon ③. Lorsque plusieurs points de jonction peuvent remplacer une même variable, plusieurs instances du greffon peuvent être créées. La manière selon laquelle les points de

jonction sont associées pour remplacer les variables des greffons est définie par le processus de combinaison de point de jonction ② en amont de la fabrique de greffon. La seconde partie du tisseur travaille uniquement avec les instances de greffon générées par la fabrique (des assemblages de composants) et a pour but de les composer. Cela consiste tout d'abord à les superposer ④, afin de produire une unique instance. Ensuite, les règles de l'instance qui interférent les unes avec les autres sont fusionnées ⑤ afin de produire l'assemblage adapté. Dans cette section, nous allons étudier en détail l'architecture du tisseur telle que présentée en FIGURE 4.16.

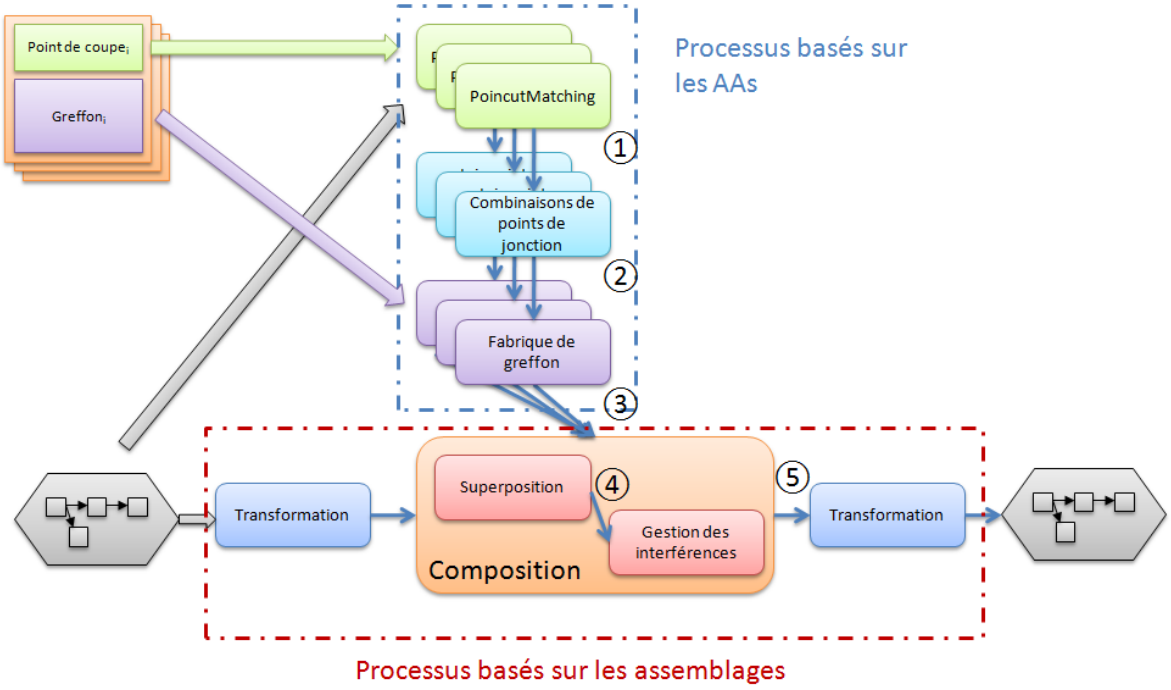


FIGURE 4.16 – Le tisseur d'Aspects d'Assemblage.

4.3.2.1 Traitement des aspects

La première étape d'un cycle de tissage consiste en la reconnaissance des points de coupe, nous parlerons du mécanisme de *Pointcut Matching* ①. Il s'agit d'une fonction qui prend en entrée la section point de coupe d'un aspect $Pointcut_i$ (une liste de règles $Rule_0, ..., Rule_j$) et la liste des ports (les points de jonctions) de l'assemblage initial $Jpoint = \{port_0 0, ..., port_n z\}$. Cette fonction produit un ensemble de listes de points de jonction $LJPoint = \{l_0, ..., l_j\}$. Chaque liste l_k contient les ports qui vérifient la règle $Rule_k$. Cette liste de points de jonction pourra ensuite être filtrée, par exemple afin d'autoriser le tissage d'AA uniquement sur les ports de certains composants ou encore pour bloquer le tissage dans certaines conditions. Nous obtenons donc après filtrage de nouvelles listes de points de jonction de cardinalité inférieure ou égale à celles qui ont été fournies en entrée du filtre.

Les éléments de ces listes seront les interfaces entre l'application et les greffons. Ainsi les greffons pour s'appliquer requièrent la présence d'un certain nombre de points de jonctions, en fait un point de jonction par règle des points de coupes. Ces règles ayant potentiellement identifiées plusieurs points de jonctions, le greffon doit pouvoir s'appliquer sur toutes les combinaisons possibles entre

les points de jonctions des différentes listes (pour rappel une liste par règle de points de coupe). Ces combinaisons sont calculées grâce au mécanisme de *JointPoint Combination* ②. Ce dernier construit à partir des listes de composants obtenues lors du pointcut matching de nouvelles listes, appelées combinaisons, dans l'ensemble *JPointComb*. Chaque combinaison est de cardinalité égale au nombre de règles se trouvant dans les points de coupe. Toute combinaison d'une autre cardinalité ne pourra être appliquée.

L'ensemble *JPointComb* peut être filtré pour éliminer certaines combinaisons de points de jonction. La liste résultante constitue le paramètre d'entrée de la fonction *Advicefactory* ③. Cette fonction utilise le greffon de l'aspect d'assemblage qui est constitué d'un ensemble de règles pour générer des instances de greffons. Ceci est réalisé en remplaçant les composants et ports décrits dans ces greffons par les points de jonction, s'il ne s'agit pas de composants instanciés par le greffon lui-même. Il s'agit donc de transformer une description abstraite d'assemblages en assemblages de composants. Ce traitement est appliqué pour chaque combinaison de points de jonction. Cette chaîne *PointcutMatching*, *JPCombination* et *AdviceFactory* sera dupliquée pour chaque AA traité par le tisseur. Ceci offre une forte modularité et permet de changer les différents algorithmes mis en place en fonction des AAs traités. Par la suite, tous les résultats de ces chaînes de traitement seront fusionnés.

Cette architecture fortement modulaire a été implémenté dans une approche à composants avec la plateforme LCA [87]. Chacune des entités décrites dans la FIGURE 4.16 correspond à un composant. De cette manière, le tisseur peut lui même être dynamiquement adapté et les chaînes correspondant au traitements réalisés sur les AAs sont générés dynamiquement (instanciation des composants correspondants) lors de la sélection d'un AA et vice versa lors de leur désélection.

4.3.2.2 Traitement des assemblages

Parallèlement à cela, un modèle de l'assemblage initial est extrait du modèle d'assemblage de composant de l'application. Ceci peut être réalisé en conservant une image de l'application initiale. En effet marquer les composants et liaisons issues d'une adaptation n'est pas suffisant puisque des liaisons peuvent avoir été réécrites lors de la fusion. Le mécanisme de composition considère donc un ensemble d'instances de greffon en vue d'en générer une unique, représentant l'assemblage initial. Nous pouvons globalement le formaliser comme suit, φ_t étant l'opération de composition à un instant t dont le premier paramètre est l'instance de greffon représentant l'assemblage initial :

$$\varphi_t(iAdvice_{00}, ..., iAdvice_{ij}) = iAdvice_{i+1\,j} \tag{4.18}$$

Il s'agit de prendre indépendamment chaque instance de greffon (chaque assemblage) et de les superposer les unes aux autres avec l'instance correspondant à l'assemblage initial. Le mécanisme de composition produira donc une seule instance de greffon G_F qui représente l'assemblage adapté. Nous pouvons identifier deux étapes dans le moteur de composition. La première a pour objectif de superposer ④ toutes les instances de greffon (des assemblages de composants) les unes avec les autres. C'est durant cette étape que peuvent apparaître des interférences entre les AAs. L'étape suivante consiste donc à résoudre ces interférences ⑤. Pour ce faire, le tisseur peut utiliser divers algorithmes. Ces algorithmes de fusion doivent garantir la cohérence de l'application tissée mais aussi la propriété de symétrie de l'opération de tissage.

4.3.3 Présentations détaillées du processus du tissage

Nous allons maintenant étudier plus en détail les différentes étapes du processus de tissage. Pour cela, nous avons introduit dans cette thèse un formalisme et une description algorithmique de ces derniers. Dans un premier temps, nous détaillerons les processus correspondant aux éléments de l'architecture du tisseur traitant des AAs. Rappelons que, dans cette dernière (FIGURE 4.16), l'ensemble de ces éléments sont dupliqués pour chaque AA. Ces processus sont donc réalisés indépendamment pour chaque AA. Ensuite, nous étudierons les éléments de l'architecture travaillant sur des assemblages de composants. Pour chacun de ces éléments, nous donnerons :

1. une brève présentation ;
2. un exemple appliqué à un scénario fil rouge ;
3. une description algorithmique ;
4. la complexité en temps de l'algorithme.

Notre scénario fil rouge est celui travaillé dans le cadre d'une application des travaux de cette thèse au bâtiment intelligent avec le CSTB (Centre Scientifique et Technique du Bâtiment). Ce scénario est à vocation pédagogique et permet d'illustrer les différentes problématiques que nous avons abordées : *« Bob rentre chez lui. Comme il fait encore jour et que le scénario d'optimisation énergétique est actif par défaut, au lieu d'allumer les éclairages artificiels, les stores sont ouverts automatiquement lorsqu'il actionne l'interrupteur des pièces où il se rend. Plus tard dans la journée, sa mère, qui sort de chez le médecin pour traiter un problème de déficience visuelle, lui rend visite et apporte au système un nouveau comportement d'assistance à la personne. Sa présence est détectée, et le système ajoute automatiquement un nouveau scénario, modifiant le comportement global de l'installation, afin de garantir un niveau d'éclairage en adéquation avec son handicap. Ainsi, lorsqu'elle actionne les interrupteurs des pièces, l'éclairage artificiel est utilisé, même si cela va à l'encontre des objectifs du scénario d'optimisation énergétique en place. »*

Pour faciliter la compréhension des mécanismes que nous allons présenter dans les sections suivantes, nous nous intéresserons à une sous-partie de ce scénario qui met en jeu le mécanisme de fusion. Nous mettrons en avant le mécanisme d'économie d'énergie permettant de contrôler l'utilisation de l'éclairage artificiel en fonction de la luminosité ambiante. Pour cela nous pouvons utiliser les deux aspects définis en FIGURES 4.13 et 5.13.

4.3.3.1 Pointcut Matching

Définition 20 : Pointcut matching
Le pointcut matching et le processus qui a pour but de déterminer dans l'assemblage de base tous les endroits où les modifications décrites dans les greffons des AAs peuvent être appliquées.

Il s'agit d'un filtre qui prend en entrées tous les ports de tous les composants présents dans l'assemblage de base (les points de jonction). Ce dernier est paramétré par les règles définies dans la section point de coupe de l'AA à traiter. Ces règles peuvent porter sur les méta-données associées aux points de jonction (nom, type ...). Ce filtre produit des listes de points de jonction, plus précisément, une liste par règle. Une liste contient les points de jonction vérifiant la règle qui lui est associée. Si une de ces listes est vide, l'AA ne peut être appliqué et le processus de tissage pour cet AA est arrêté. Le type des points de jonction (fourni ou requis) constituant ces listes est connu. Chaque liste ne

contient qu'un seul type de port.

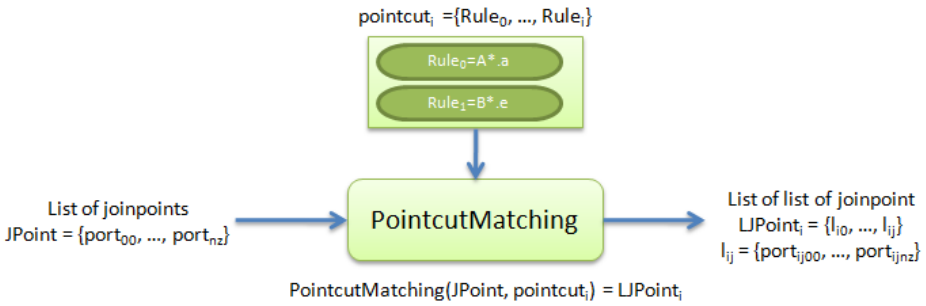


FIGURE 4.17 – Pointcut Matching.

Ainsi, dans un premier temps, le tisseur extrait tous les ports de l'assemblage de base et construit l'ensemble des points de jonction appelé : *JPoint*. Ensuite, il récupère toutes les sections pointcut des AAs. Pour chaque AA, le processus de *pointcut matching* associé évalue l'ensemble des règles sur l'ensemble des ports de l'assemblage afin de construire des listes de *component.port* (une liste par règle, l'ensemble des points de jonction de la liste vérifiant la règle associée). L'ensemble de ces listes est noté : *LJPoint*. La FIGURE 4.19 présente un des points de coupe de notre scénario. Nous considérons un assemblage initial composé d'une lampe, de deux switch, d'un téléphone et d'un capteur de luminosités. Le point de coupe donné au mécanisme de pointcut matching et celui présenté en FIGURE 4.18. Cet exemple de point de coupe est basé sur un matching syntaxique sur les méta-données des points de jonction. Le langage utilisé est AWK.

```
1 sender:=/switch[[:digit:]].^on/
  receiver:=/light[[:digit:]].SetStatus/
  condition:=/*.^Brightness_Value_Evented_NewValue/
```

FIGURE 4.18 – Point de coupe d'un aspect d'assemblage.

Le résultat du pointcut matching se compose alors d'une liste contenant trois listes de points de jonction. La première contient un port *light.SetStatus*, la seconde un port *brightness1.^ Brightness_Value_Evented_NewValue* et enfin la dernière contient deux ports *on* des composants *switch1* et *switch2*. Ce processus peut être défini plus formellement comme dans l'algorithme 1 dont la complexité est de :

$$O(j \times card(JPoint) \times cout\ de\ la\ fonction\ satisfy)$$

L'évaluation d'un point de jonction pour une règle dépend de la façon dont nous avons défini nos règles. Un exemple d'implémentation est la définition de ces dernières en utilisant des expressions régulières. Dans ce cas, la complexité de l'évaluation d'une règle peut être de $O(ml)$ avec m=taille de l'expression et l=taille du mot.

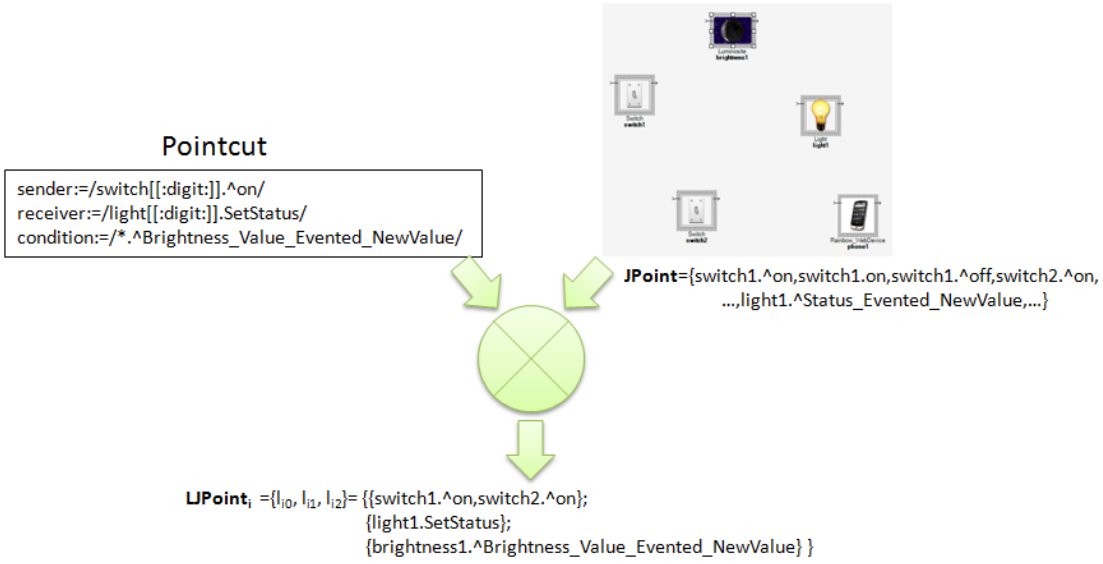


FIGURE 4.19 – Exemple de Pointcut Matching.

4.3.3.2 Combinaisons de points de jonction

Définition 21 : Fonction de combinaison de points de jonction
Le processus de combinaison des points de jonction, aussi appelé (JoinPoint Combination) a pour objectif de combiner les points de jonction vérifiant le pointcut matching selon une politique associée à l'AA dans l'optique de définir *où* et *comment* doivent être dupliqués les AAs.

Les listes données en entrée à ce processus sont les listes des points de jonction créées par le pointcut matching. Les greffons pour s'appliquer requièrent un élément de chaque liste : une combinaison. Le greffon est alors appliqué autant de fois qu'il y a de combinaisons de points de jonction. Ceci permet de tirer parti des avantages de l'AOP en offrant une forte réutilisabilité et en minimisant la dispersion du code. Ce mécanisme de combinaison de points de jonction peut s'écrire comme une fonction qui construit à partir de *LJPoint* un ensemble de combinaisons $JPointComb = \{Combi_0, ..., Combi_j\}$.

Définition 22 : Combinaison de points de jonction
Une combinaison de point de jonction est une liste de point de jonction ayant vérifié le pointcut matching. La cardinalité d'une combinaison est égale au nombre de règles se trouvant dans le pointcut de l'AA, chaque point de jonction dans une combinaison ayant vérifié une règle de point de coupe différente.

Il est important de noter que divers algorithmes de combinaisons peuvent être implémentés. Il n'est pas nécessaire de calculer toutes les combinaisons possibles entre les points de jonction de chaque liste. Ainsi, par exemple, nous pouvons définir un algorithme combinant les points de jonction en fonction de leur nom ou de méta-données qui leur sont associées. De tels algorithmes peuvent être moins coûteux en temps. D'autre part, puisque dans le tisseur les traitements portant sur les AAs (étapes ①, ②, ③) sont dupliqués pour chaque AA, il est possible de choisir et d'associer à chaque

Algorithm 1 $PointcutMatching(JPoint, PointCut_i)$

l_{ij} : a list of ports (joinpoints) where $l_{ij} = port_{ij00}, ..., port_{ijnz}$ and j is the number of lists, which is equal to the number of rules in $PointCut_i$
$LJPoint_i$: a set of joinpoint lists where $LJPoint_i = \{l_{io}, ..., l_{ij}\}$
$JPoint$: the set of ports from the base assembly $port_{00}, ..., port_{nz}$

create $LJPoint_i$
for $s = 0$ to j **do**
  Add a new list l_{is} to $LJPoint_i$
  for $t = 0$ to card(JPoint) **do**
    if JPoint[t] satisfy the rule $Rule_{is}$ **then**
      Add JPoint[t] to the list l_{is}
    end if
  end for
end for

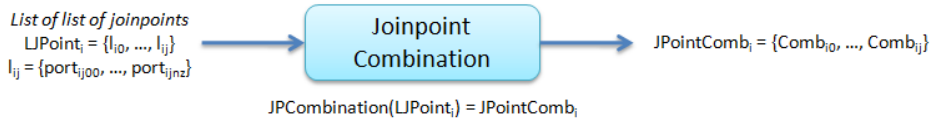


FIGURE 4.20 – Combinaisons de points de jonction.

AA un algorithme différent. De cette manière, un designer, peut gérer une partie de la portée d'un AA en lui associant un algorithme pertinent. La FIGURE 4.21 présente un exemple de combinaison de points de jonction. L'algorithme choisi consiste à calculer toutes les combinaisons possibles. Les données fournies en entrée de la fonction de combinaison dans cet exemple sont celles obtenues dans l'exemple précédent, c'est-à-dire les points de jonctions permettant l'application de l'AA. Le nombre de combinaisons possible est de deux, puisque deux points de jonction ont étés associés la variable *sender*. L'algorithme 2 est celui utilisé dans l'exemple, sa complexité est la suivante :

$$O(card(JPoint)^j).$$

Comme nous l'avons vu cette complexité peut varier et être plus faible suivant la politique de combinaison choisie.

4.3.3.3 Fabrique de greffons

Définition 23 : Fabrique de greffons
L'objectif de la fabrique de greffon est de construire, à partir de la liste de combinaisons de point de jonction, des instances de greffon. Elle permet de relier les greffons qui sont des représentations abstraites d'assemblages de composants à l'assemblage de base et ainsi de réaliser autant d'instances de greffon qu'il y a de combinaisons.

La fabrication d'une instance de greffon consiste à remplacer les variables des règles d'un greffon par les points de jonction de la combinaison traitée. La fabrique de greffon produit donc une liste d'instances de greffon $iAdviceList_i$. Ainsi, la règle du greffon de la FIGURE 5.13 :

Algorithm 2 JPCombination(*LJPoint*)

ACombination : list of joinpoint
Product : Integer : number of possible combination
mult : Integer : number of combination using the joinpoint
lcomb : list of combination

mult=1 ;
create JPointComb
for $i = 0$ to $card(LJPoint)$ **do**
  Create lcomb
  ACombination.Clean
  $product = product/(card(LJPoint[i]) - 1)$
  for $j = 1$ to $card(LJPoint[i])$ **do**
    for $k = 0$ to $product$ **do**
      ACombination.Add($LJPoint[i][j]$)
    end for
  end for
  for $j = 1$ to $mult$ **do**
    lcomb.Add(ACombination)
  end for
  JPointComb[i]= lcomb
  $mult = mult \times (card(LJPoint[i]) - 1)$
end for
return JPointComb

sender.^Status_Evented_NewValue -> (receiver.SetStatus) est remplacée dans une instance par *switch1.^Status_Evented_NewValue -> (light1.SetStatus)*.

Tout comme pour le pointcut matching et les combinaisons de points de jonction, à chaque AA est associé une fabrique d'instance de greffon. De cette manière $iAdviceList_i$ est uniquement composée d'instances d'un même greffon. Ce processus peut être décrit plus formellement comme dans l'algorithme 3. Puisque la complexité de la fonction *replace* peut être constant $O(1)$, la complexité de cet algorithme est la suivante :

$$O(k \times w)$$

Algorithm 3 AdviceFactory($JPointComb_i$)

k : number of combination
w : number of advice rules

for $s = 0$ to k **do**
  for $t = 0$ to w **do**
    Replace variable from $ARule[t]$ using $JPointComb[s]$
  end for
end for

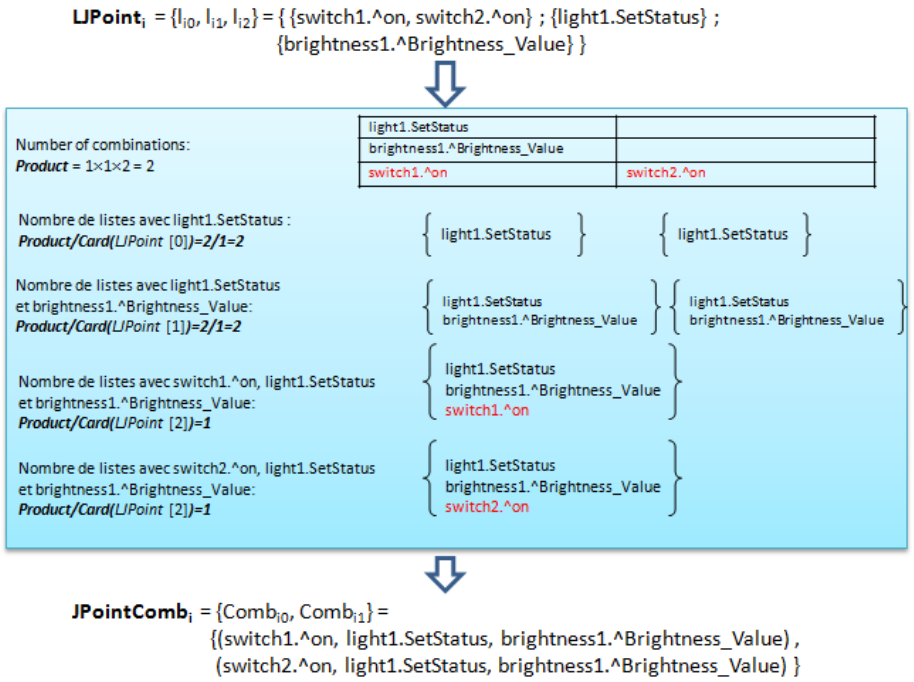


FIGURE 4.21 – JoinPoint Combination Example

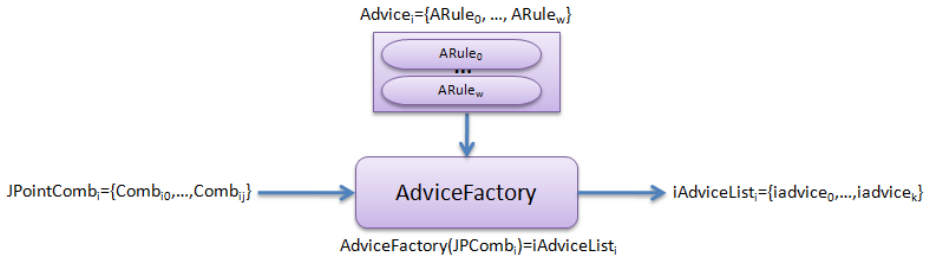


FIGURE 4.22 – Fabrique de greffons.

4.3.3.4 Composition

Le mécanisme de composition considère donc un ensemble d'instances de greffon en vue d'en générer un unique, représentant l'assemblage final. Ceci permet de construire un unique greffon G_T qui contiendra toutes les modifications à apporter dans l'assemblage initial. Cette phase de composition se décompose elle-même en trois étapes. La première, de superposition de toutes les instances de greffons, a pour objectif de produire une unique instance de greffon. Cette instance, qui comporte un ensemble de règles visant à construire un nouvel assemblage de composants, peut alors contenir des règles en interaction les unes avec les autres. La seconde étape de la composition consiste donc à identifier ces interactions. Enfin, c'est sur ces points d'interaction que le mécanisme de fusion interviendra afin de produire un assemblage final sans interférences.

4.3.3.5 Superposition

Définition 24 : Superposition
La superposition d'assemblages de composants est une expression qui construit systématiquement un unique assemblage à partir de plusieurs assemblages de composants intermédiaires.

La fonction Superposition prend en entrée les instances de greffon produites par les fabriques de greffons de chaque AA : *iAdviceList* et l'instance de greffon G_0 correspondante à l'assemblage initial. Elle produit en sortie le greffon $G_T = \{GRule_0, ..., GRule_a\}$ qui regroupe toutes les règles des instances de greffons qui produiront l'assemblage final. C'est à partir de cette étape du processus de tissage que des interactions entre règles d'adaptation et ainsi entre AAs peuvent apparaître dans G_T.

La FIGURE 4.23 présente un exemple de superposition. Dans cet exemple, les entrées du mécanisme de superposition sont les deux instances de greffons obtenues grâce à l'aspect de contrôle de la luminosité ainsi que deux instances obtenues grâce à l'aspect permettant d'allumer la lampe grâce aux switchs. Les cinq premières règles de la nouvelle instance d'advice G_0 représentent l'assemblage de base avant superposition. Les règles suivantes sont ajoutées par superposition. Nous noterons que les variables présentes dans les advices des AA ont bien étés remplacées dans leur instances par des points de jonctions. Ce processus peut être décrit plus formellement comme dans l'algorithme 4.

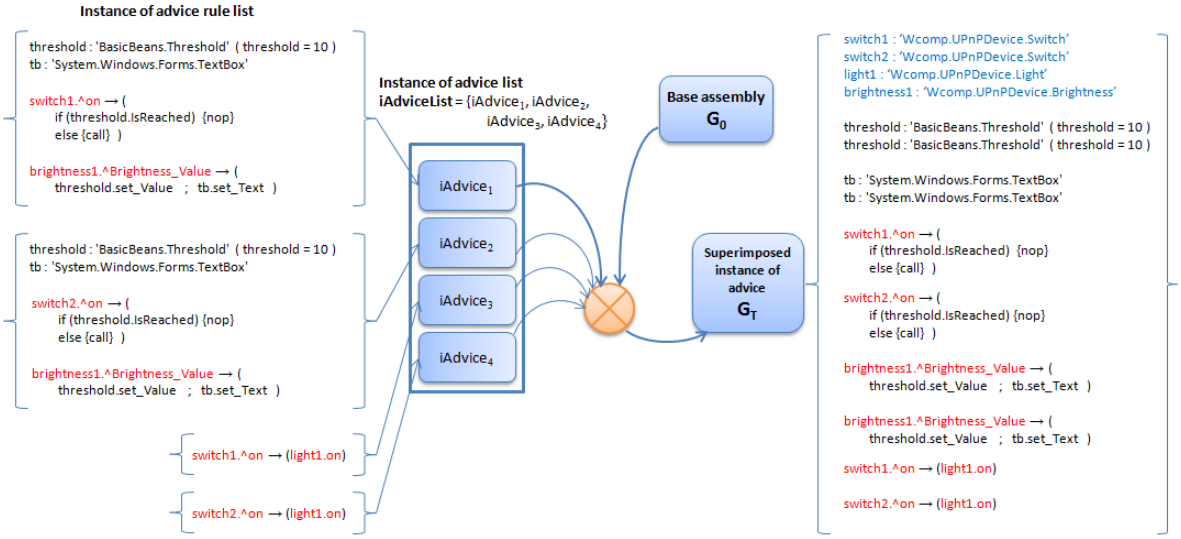


FIGURE 4.23 – Exemple de superposition

La complexité de cet algorithme est de : $O(y \times max(Card(iAdvise_i)) \times card(G_0))$.

Algorithm 4 Superimpose(iAdviceList)

y : number of instance of advice

for $d = 0$ to y **do**
  for $t = 0$ to $card(iAdvise_d)$ **do**
    if $iAdvise_d[t] NotInG_0$ **then**
      Add $iAdvise_d[t]$ to G_0
    end if
  end for
end for

4.3.3.6 Fusion

Définition 25 : Fusion
Le mécanisme de résolution des conflits a pour but de gérer les interactions qui peuvent intervenir lorsque plusieurs instances de greffon sont tissées sur un même point de jonction (shared joinpoint) tout en conservant la consistance de l'application et en garantissant la propriété de symétrie de l'opération de tissage.

C'est à partir de ce composant noté $\otimes$ que le mécanisme de gestion des interférences construit sa solution. Il permet d'identifier où se trouvent les interférences.

Le mécanisme de fusion a pour but de remplacer ces composants avec des composants de sémantique connue. Grâce à ce mécanisme, nous pouvons garantir la consistance du comportement de l'application. Pour ce faire, diverses techniques peuvent être utilisées, par exemple basées sur les graphes ou les langages. Basiquement, l'algorithme pour résoudre ces interférences parcourt l'ensemble de tous ces composants dans l'objectif de faire tourner sur chacun d'eux le moteur de fusion. Ce processus peut être décrit plus formellement comme dans l'algorithme 5.

Algorithm 5 InterferenceResolution(iAdvice)

for $s = 0$ to $card(List\otimes)$ **do**
  $Merge(List \otimes [s])$
end for

Un de nos mécanismes de résolution des conflits, dans le cadre de l'informatique ambiante, est basé sur le langage ISL4WComp [64].

Mécanisme de fusion basé sur ISL4WComp Le mécanisme de fusion d'ISL4WComp est basé sur les opérateurs listés dans le tableau 4.3. Les règles d'interaction sont exprimées sous la forme d'arbres sémantiques. Son membre gauche en est la racine, les opérateurs du langage les noeuds et les points de jonction ou les ports des composants boite noire les feuilles. Fusionner deux arbres consiste à fusionner les noeuds des arbres en partant de la racine, puis à propager cette fusion dans leurs branches. La fusion des noeuds (opérateurs) repose sur des règles prédéfinies comme celles présentées en FIGURE 4.24. Vingt quatre règles de fusion sont définies dans [64]. La fusion de chacun de ces opérateurs a été définie comme symétrique [64, 114]. La fusion de deux opérateurs peut être décrite

avec plusieurs règles. Par exemple, l'opération de fusion de deux opérateurs *if* est basée sur deux règles. Écrivons $\otimes$ l'opération de fusion, $if(condition1, thenA, elseB) \otimes if(condition2, thenC, elseD)$ est égale à :

- *Si* $condition1 = condition2$:
  $if(condition1, thenA \otimes thenC, elseB \otimes elseD)$
- *Si* $condition1 \neq condition2$:
  $if(condition1, if(condition2, thenA \otimes thenC, thenA \otimes elseD), if(condition2, elseB \otimes thenC, elseB \otimes elseD))$

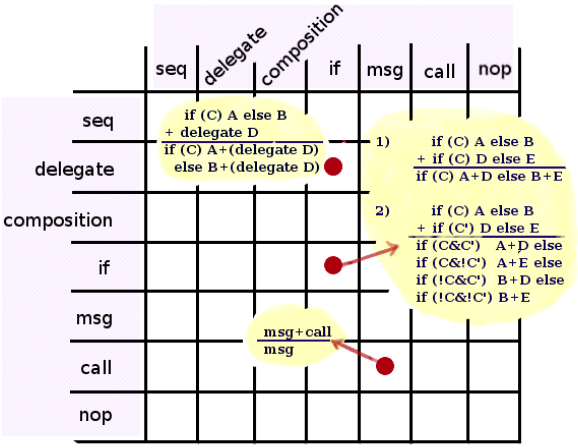


FIGURE 4.24 – Matrice de fusion des opérateurs [114].

Lorsque deux règles qui ajoutent deux liaisons n'utilisent pas d'opérateur du langage et sont en conflit, l'opération de fusion consiste à ajouter un opérateur *parallel* entre les deux liaisons. Ceci garantit également la propriété de symétrie de l'opération de fusion. Une fois que les arbres sont fusionnés, ils sont transformés en instructions élémentaires (add/remove component/binding), et les opérateurs sont alors représentés dans l'assemblage de composants par des composants de sémantique connue. Ainsi par exemple, lorsque les deux AAs présentés dans la SECTION 4.3.1.4 sont composés, une interaction apparaît sur le port *switch.^on*. Le résultat de l'opération de fusion des deux règles en interaction est décrit en FIGURE 4.25.

Dans cet exemple, nous pouvons voir la propagation de l'opération de fusion jusqu'aux feuilles. En premier lieu, elle fusionne le message *light.on* et l'opérateur *if*. Ensuite, elle fusionne le message *light.on* et l'opérateur *nop* dans la branche then du *if* et le message *light.on* et l'opérateur *call* dans sa branche else. L'opérateur *nop* étant absorbant, le résultat de la fusion de la branche then est *nop*. A l'inverse, dans la branche else, l'opérateur *call* est neutre. Le résultat de la fusion pour cette branche est donc *light.on*.

Grâce à ce mécanisme de fusion, les AAs garantissent que la composition des instances de greffons est symétrique. Il s'agissait d'un des pré-requis pour que l'opération de tissage d'aspects soit symétrique. Nous allons dans la section suivante résumer l'ensemble des propriétés des AAs qui garantissent que leur opération de tissage est symétrique.

Rule 1:
switch.^on->light.on

Rule 2:
switch.^On -> (
if (threshold.IsReached) {nop}
else {call})

Merging result:
Switch.^on->light.on + if(threshold.IsReached){nop}else{call} (Step 1)
->if(threshold.IsReached) light.On + nop else light.On + call (Step 2)
->if(threshold.IsReached) nop else light.On (Step 3)

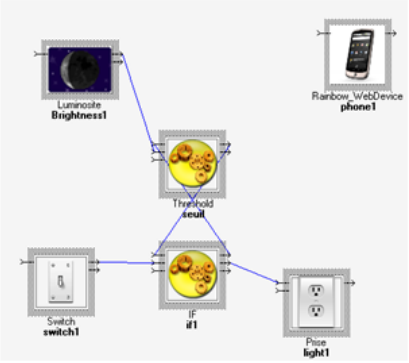


FIGURE 4.25 – Résultat de la fusion.

4.3.4 Propriétés logiques

Comme nous l'avons déjà vu, la propriété de symétrie se décompose elle-même en trois sous-propriétés : associativité, commutativité et idempotence. Nous avons définis en SECTION 4.2.3.2 une liste de caractéristiques que des aspects doivent respecter afin que leur opération de tissage soit symétrique. Dans le cadre des AAs, cette liste est vérifiée.

L'idempotence spécifie qu'un même AA ne peut être tissé qu'une fois. Un AA est identifié par son nom dans un espace de nommage. Si deux AAs sont composés de règles identiques mais de nom différent, ils seront tous deux appliqués. Cette contrainte des espaces de nommage a des conséquences fortes sur cette propriété d'idempotence, en particulier sur les AAs qui instancient des composants boite noire. En fonction de ces contraintes, afin de garantir l'idempotence, la politique suivante est appliquée :

- Si n AAs différents possèdent des règles instanciant un composant de même type, alors celui-ci sera instancié n fois.
- Si le même AA contenant une règle pour instancier un composant boite noire est déployé n fois, alors le composant boite noire est instancié 1 fois.

La commutativité permet de garantir que l'ordre dans lequel sont tissés les AAs n'a pas d'importance. Cette propriété est garantie grâce à l'utilisation d'opérateurs de sémantique connue permettant de fusionner plusieurs règles. Par construction, il est défini que la fusion de ces opérateur est commutative. Cette propriété a été prouvée sur la fusion, deux à deux, de chacun des opérateurs [64]. Ajouter un nouvel opérateur impliquera de prouver cette propriété avec tous les autres opérateurs. Lorsque deux règles sont en interaction (elles mettent en jeu un même point de jonction), si elles incorporent des opérateurs, alors elles sont fusionnées en fonction de ces derniers. Leur fusion est donc commutative. Si aucun opérateur n'est utilisé dans les règles alors un opérateur par défaut peut être introduit comme par exemple un opérateur de parallélisme. D'autre part, pour garantir cette propriété un AA ne peut pas utiliser dans son pointcut un composant instancié par un autre AA. De la même manière, un AA n'a pas la capacité pas supprimer de composants.

Enfin, la propriété d'associativité permet de regrouper des compositions d'AAs dans des représentations plus abstraites : $(AA_1 \otimes AA_2) \otimes AA_3 = X \otimes AA_3 | X = (AA_1 \otimes AA_2)$. Cette propriété est également garantie grâce aux opérateurs des advices. Classiquement, l'opération de tissage d'aspects est uniquement associative à droite [16].

Définition 26 : Propriété de symétrie des AAs
$AA_1 \otimes AA_2 = AA_2 \otimes AA_1$ (commutativité)
$(AA_1 \otimes AA_2) \otimes AA_3 = AA_2 \otimes (AA_1 \otimes AA_3)$ (associativité)
$AA_1 \otimes AA_1 = AA_1$ (idempotence)

Grâce à cette symétrie l'opération de tissage est déterministe, c'est-à-dire que pour un set d'AAs et un assemblage initial donné, le résultat de l'opération de tissage sera toujours le même quelque soit l'ordre dans lequel les AAs sont tissés (EQUATION 4.8). Les AAs sont donc un mécanisme adapté pour la réalisation de cascades puisqu'ils respectent les contraintes précédemment évoquées.

4.4 Les Cascades d'Aspects d'Assemblage

Dans cette section nous proposons d'utiliser les Aspects d'Assemblages dans notre approche de cascades d'aspects. En effet, ces derniers possèdent bien les propriétés logiques nécessaires. Nous étudierons les cascades d'AAs que nous appliquerons à notre scénario pour illustrer les avantages des cascades d'AAs en terme de combinatoire de règles d'adaptation à écrire et de configurations atteignable par combinaisons d'AAs.

4.4.1 Principes

Les cascades d'Aspects d'Assemblage sont donc des ensembles ordonnés d'ensemble d'AAs. Nous pouvons donc réécrire l'ÉQUATION 4.19 comme :

$$\begin{aligned} &T_t(Ass_0, CAA) = Ass_{final} \mid CAA = \{CAA_0, ..., CAA_n\} \\ &CAA_n = \{AA_{00}, ..., AA_{0m}\} \end{aligned} \tag{4.19}$$

Une cascade se compose donc de divers ensembles. Ces ensembles, appelés *FunctionalSet* dans le méta-modèle d'une cascade d'AAs présenté en FIGURE 4.26, sont ordonnés. Un *FunctionalSet* se compose d'un ensemble d'AAs. A une cascade peuvent être associés un nom et un espace de nommage. Tous les AAs appartenant à cette cascade, s'ils ne déclarent pas eux même un espace de nommage particulier, appartiennent à ce dernier. Un AA peut donc également déclarer son espace de nommage. Le nom complet d'un AA se compose donc de l'espace de nommage de la cascade, le nom du *FunctionalSet* et le nom de l'AA. De cette manière, deux AAs, ayant des noms de base identiques, déployés dans deux groupes fonctionnels différents, n'ont donc pas le même nom. Comme pour les AAs classiques, s'ils ont des règles identiques, alors celles-ci sont dupliquées. Plusieurs cascades peuvent également partager le même espace de nommage. Le nombre d'AAs et de groupe fonctionnels dans une cascade n'est pas limité. Lors du déploiement à un groupe fonctionnel correspondra un dépôt d'Aspects d'Assemblages.

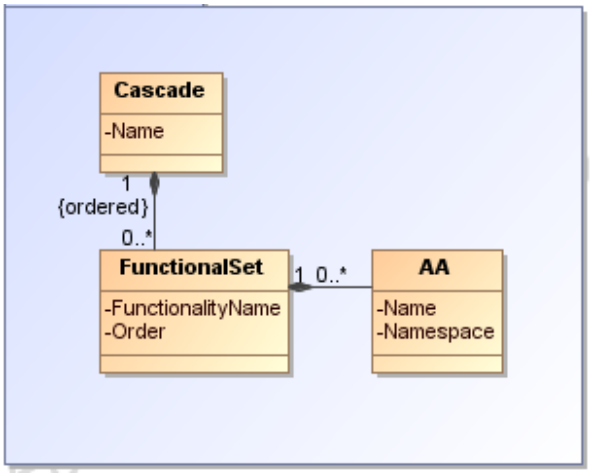


FIGURE 4.26 – Méta-modèle de Cascade d'AAs.

4.4.2 Combinaisons d'AAs et décomposition fonctionnelle

Afin d'illustrer les cascades d'AAs, nous allons maintenant utiliser l'ensemble du scénario défini en SECTION 4.3.3. Nous complétons l'exemple précédent afin de mettre en place l'ensemble du scénario fil rouge. L'exemple précédent mettait en place le comportement d'économie d'énergie consistant à autoriser l'allumage des lumières uniquement lorsque la luminosité était suffisamment faible. Désormais, si la luminosité extérieure est suffisamment élevée alors les stores sont ouverts au lieu d'allumer la lumière. Nous y ajoutons le second comportement d'assistance à la personne qui, à l'aide d'un lecteur RFid, identifie si la mère de Bob est dans la pièce afin d'utiliser uniquement l'éclairage artificiel. Dans l'optique de maximiser le nombre de configurations atteignables par le système tout en limitant le nombre de règles de reconfiguration à écrire, la décomposition peut se faire comme décrite en FIGURE 4.27. Puisque dans le scénario nous pouvons identifier deux comportements, nous définirons deux cascades différentes : (1) assistance à la personne et (2) économie d'énergie.

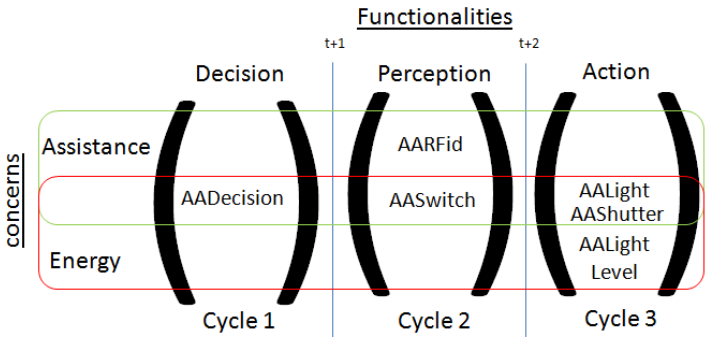


FIGURE 4.27 – Cascaded AA.

Dans un premier temps, observons le comportement d'assistance à la personne qui est prioritaire. Ce comportement nécessite la définition d'une cascade d'AAs sur trois cycles de tissage. Nous écrirons dans un premier temps un AA (FIGURE 4.29) pour le premier cycle de tissage. Il s'agit de la

fonctionnalité de décision du système. Elle fera la liaison entre les parties de perception et d'action du système. Par conséquent, elle sera fortement réutilisée par les autres fonctionnalités du système et variera peu. Nous aurions pu déployer la fonctionnalité de perception en premier de manière à ce que le mécanisme de décision soit créé en fonction du mécanisme de perception existant. Cependant, pour ce scénario, cela aurait nécessité la réécriture plusieurs fois de la partie point de coupe de l'aspect AAdecision en fonction de la perception nécessaire à son application. Rappelons que dans l'optique de minimiser le nombre de règles à écrire tout en maximisant le nombre de configurations atteignable par le système, les fonctionnalités les plus réutilisées par les fonctionnalités suivantes sont placées dans les premiers cycles. Ici le mécanisme de décision est fortement réutilisé, il est donc placé dans le premier cycle. La FIGURE 4.28 présente le graphe des dépendances entre les différents aspects des cascades.

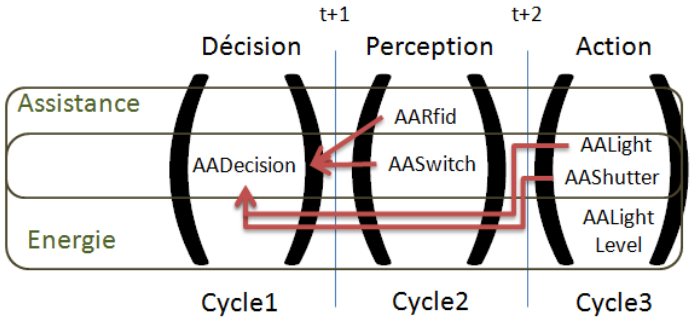


FIGURE 4.28 – Dépendances entre les aspects.

L'objectif de l'AA de décision est d'instancier un timer et un composant de décision dont le rôle est d'indiquer s'il est faut allumer la lampe ou ouvrir le store en fonction d'un identifiant et d'une heure passés en entrée.

```
Advice:
  schema dec():
        Decision : 'WComp.BasicBeans.DecisionEntity';
        Timer : 'WComp.BasicBeans.Timer';

        Timer.^Status _New_Evented_Value -> (Decision.SetTime)
```

FIGURE 4.29 – Aspect de décision.

Ensuite, deux AAs pour un second cycle de tissage permettent de décrire le mécanisme de perception qui devra être implémenté. Ils permettront, de manière indépendante, de connecter le lecteur RFiD (FIGURE 4.30) et l'interrupteur au composant de décision (FIGURE 4.31). Lorsqu'un tag RFiD est lu par un lecteur ou lorsque l'interrupteur change d'état, le mécanisme de décision en sera informé. Ces deux AAs pour s'appliquer requièrent donc la présence du composant de décision instancié lors du premier cycle de tissage.

```
Pointcut
  rfid:=/rfid.*/
  DecisionEntity:=/Decision[[:digit:]]/
Advice:
  schema obs(DecisionEntity,rfid):
      Rfid.^ value_Evented_NewValue->(
          DecisionEntity.Manage)
```

FIGURE 4.30 – AA de perception pour le Rfid.

```
Pointcut:
  switch:=/switch.*/
  DecisionEntity:=/Decision[[:digit:]]/
Advice:
  schema obs(DecisionEntity,switch):
      switch.^ value_Evented_NewValue->(
          DecisionEntity.Manage)
```

FIGURE 4.31 – AA de perception pour l'interrupteur.

Enfin, nous ajoutons des AAs pour connecter le composant de décision aux lampes (FIGURE 4.33) et aux stores (FIGURE 4.32). Ces AAs sont destinés au troisième cycle de tissage. Le système peut donc être fonctionnel même en l'absence d'un des actionneurs ou capteur.

```
Pointcut:
  Shutter:=/Shutter.*/
  DecisionEntity:=/Decision[[:digit:]]/
Advice:
  schema action(DecisionEntity,Shutter):
      DecisionEntity.^ ShutterManagementEv
      ->(Shutter.SetState)
```

FIGURE 4.32 – AA pour la gestion des stores.

```
Pointcut:
  light:=/light [[:digit:]]/
  DecisionEntity:=/Decision[[:digit:]]/
Advice:
  schema ActionLight(light, DecisionEntity):
      DecisionEntity.^ LightManagementEv
          ->(Light.SetState)
```

FIGURE 4.33 – AA pour la gestion des lampes.

Si nous considérons maintenant le comportement d'économie d'énergie, alors la cascade qui permet de le décrire, peut réutiliser les AAs pour la perception et la décision utilisées par la première. Nous ajoutons, donc pour le troisième cycle de tissage, un AA similaire à celui présenté précédemment en FIGURE 4.13 ajoutant un filtre sur un appel à l'ouverture de la lampe qui permet de rediriger ces appels vers le store en fonction de la luminosité extérieur. A partir de la formule présentée en FIGURE 4.9, nous pouvons déduire que ces AAs peuvent être combinés afin d'autoriser la production de $2^2 \times 2^3 = 32$ configurations. En effet, une fois l'AA de décision tissé, tous les autres AAs peuvent être appliqués indépendemment les uns des autres.

Étudions maintenant une mise en œuvre du processus de tissage dans notre scénario. Considérons un assemblage initial composé d'un représentant vers un service pour dispositif d'un interrupteur, un représentant vers un dispositif lampe, un représentant vers un store et enfin un représentant vers un lecteur Rfid. Considérons que la cascade du comportement d'économie d'énergie est déployée. Dans un premier temps l'AA *AADecision* est tissé dans un premier cycle de tissage. Le composant décision est maintenant dans l'assemblage. L'AA pour le second cycle *AASwitch* est ensuite tissé. De la même manière, deux AAs du troisième cycle peuvent être tissés : *AAlight* et *AAshutter*. L'assemblage résultant est présenté en FIGURE **??**.

Par la suite, un représentant vers un capteur de luminosité qui vient d'être installé est instancié dans l'assemblage. Cette apparition déclenche une nouvelle opération de tissage des cascades, l'aspect *Brightness_light* présenté en FIGURE 4.13 est tissé. Par la suite, la grand mère de Bob entre dans la maison et déploie la cascade pour le comportement d'assistance à la personne. Désormais, avec les

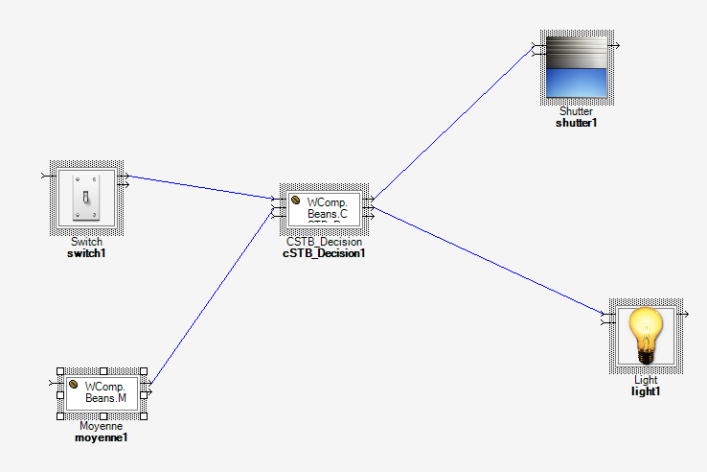


FIGURE 4.34 – Assemblage avant l'installation d'un capteur de luminosité.

autres AAs, l'aspect *AARFid* peut être tissé. L'assemblage résultant est présenté en FIGURE **??**

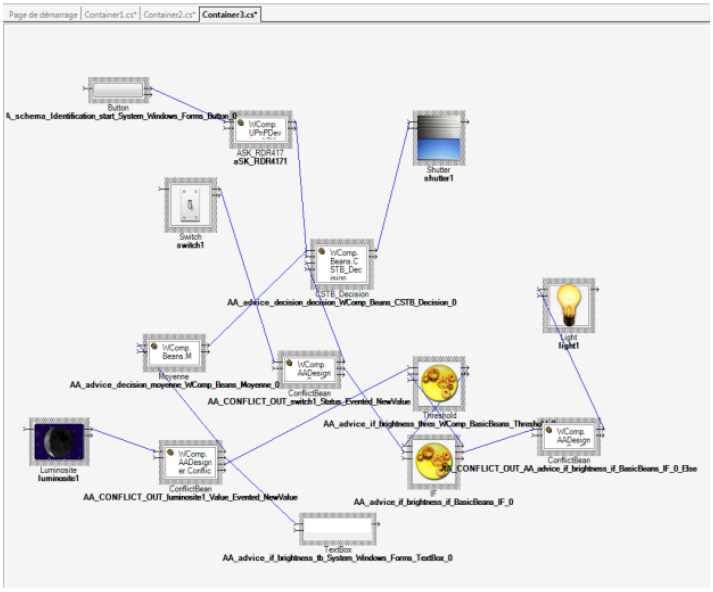


FIGURE 4.35 – Assemblage après l'ajout du comportement d'assistance.

4.4.2.1 Gestion de la portée

Nous avons vu que dans le tisseur d'Aspects d'Assemblage, les étapes traitant des aspects d'assemblage (1),(2) et (3) sont propres à chaque AA déployé. La politique selon laquelle les points de coupe et les combinaisons de points de jonction sont calculés peut donc changer d'un AA à un autre, personnalisant la façon dont est calculée la portée de l'AA. Grâce aux capacités de décomposition offertes par les cascades, les AAs qui les composent peuvent avoir une granularité plus fine, c'est-à-dire que tous les aspects d'une cascade peuvent ne contenir qu'une règle d'adaptation. Des politiques plus fines de gestion de la portée des AA peuvent donc être appliquées.

Nous pouvons voir l'intérêt de cette gestion avec les exemples de cascades que nous venons de présenter. Par exemple, dans l'optique de connecter plusieurs capteurs de luminosité à un unique composant permettant d'agréger leurs données, la politique de combinaisons des points de jonction doit être de type $n : n$. Cela signifie, que pour tout point de jonction identifié, il faut calculer toutes les combinaisons possibles avec les autres points de jonctions ayant vérifié le pointcut matching. Ceci permet d'appliquer le greffon le plus possible. Inversement, lorsque nous utilisons le lecteur RFid, nous choisirons plutôt une politique de type 1 : 1, c'est-à-dire que chaque point de jonction n'apparaît que dans une combinaison (par défaut pour cela les points de jonction sont triés en fonction de leurs noms, d'autres méta-données pourraient être utilisées). Dans l'exemple, il y a seulement un composant de décision, et par conséquent, seulement un lecteur RFid lui sera associé. La FIGURE 4.36 présente ces politiques. Pour associer une politique de combinaison à un aspect, il suffit lors de son déploiement d'indiquer la politique qui lui est associé. En fonction de l'indicateur fourni au déploiement de l'aspect, le composant permettant de calculer les combinaisons de points de jonction sera déployé dans le tisseur dans la chaine de traitement de cet aspect. Actuellement, la création de nouvelle politique de combinaison passe par la définition de nouveaux composants pour le tisseur. Il est également possible d'envisager des composants génériques qui pourraient être paramétrés par des règles.

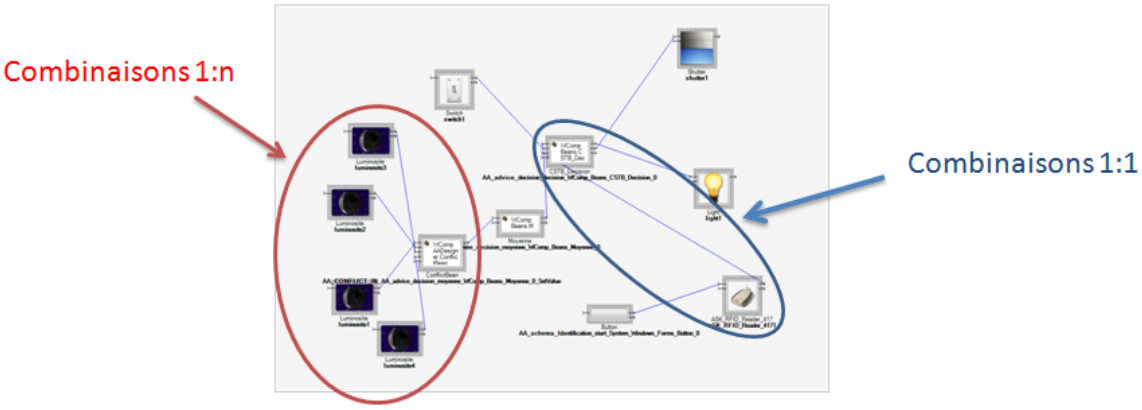


FIGURE 4.36 – Différentes portées de tissage.

Les principales modifications apportées au tisseur interviennent dans la gestion des représentations de l'application initiale, de l'application courante, et de l'application résultant de l'adaptation entre les cycles de tissage et la gestion des cycles. De fait, ces modifications impactent les mécanismes de synchronisation entre le modèle de l'application et l'application s'exécutant. Nous allons maintenant voir plus en détail de quelle manière les cascades sont appliquées sur des représentation abstraites de l'application avant que les modifications en soient reportées dans l'application s'exécutant. Nous verrons que, pour limiter les modifications apportées à l'application, c'est seulement une fois tous les cycles de tissages réalisés que l'application est adaptée.

4.4.3 Des models@runtime pour minimiser les modifications dans l'application s'exécutant

Maintenant que nous avons étudié de quelle manière sont calculées les adaptations qui doivent être portées sur l'application, nous allons détailler de quelle manière elles y sont reportées. Comme nous l'avons mis en avant dans l'état de l'art (SECTION 2.3), afin de limiter le nombre de modifications réalisées dans l'application, nous utilisons des modèles de l'application à l'exécution sur lesquels sont effectuées l'ensemble des modifications durant les différents cycles de tissage avant d'être reportées dans l'application s'exécutant comme un ensemble d'instructions élémentaires de modification de l'assemblage. Pour cela, nous présenterons dans un premier temps notre méta-modèle d'assemblage de composants dont une instance est utilisée à l'exécution pour représenter l'application s'exécutant. Ce méta-modèle n'a pas pour objectif d'être appliqué à l'ensemble des plateformes d'exécution à base de composants. Il ne s'agit ni de l'union des méta-modèles des plateformes existantes ni d'un méta-modèle contenant l'ensemble des entités nécessaire et suffisant pour exprimer tout type d'assemblage de composants. D'autres travaux se sont interessés à cette problématique, comme par exemple, le méta-modèle d'architecture proposé dans [96] qui résulte d'une analyse de modèles comme UML, SCA et de plateformes comme OSGi, Fractal, OpenCom... ou encore dans le domaine des ADL (Architecture Description Language) [137] comme le montre l'étude suivante [149]. Nous verrons ensuite l'ensemble des transformations que le processus de tissage met en jeu pour identifier parmi celles-ci les processus de synchronisation que nous détaillerons plus particulièrement.

4.4.3.1 Méta-modèle d'application

Notre modèle d'assemblage de composants prend la forme d'un graphe conforme au méta-modèle décrit en FIGURE 4.37. Un nœud du graphe est un port de l'assemblage de composant qui peut avoir des paramètres. Nous noterons la présence de deux types de nœuds : les nœuds boite noire et les nœuds boite blanche dont la sémantique est connue. Ce sont les noeuds qui nous permettront de gérer la fusion de plusieurs modèles. Les arcs de ces graphes sont les liaisons entre ports d'entrée et ports de sortie. Aux ports et liaisons peuvent être associés des méta-données. Ces méta-données peuvent être par exemple le fait qu'une liaison est *optional* ou *mandatory*. Un assemblage peut être formalisé comme dans l'EQUATION 4.20. Un assemblage contient un ensemble de ports $Jpoint$, qui sont les points de jonction de nos aspects, et un ensemble de liaisons L. A chaque port sont associés deux indices : (1) le premier indique le composant auquel appartient le port et (2) le second identifie le port parmi ceux du composant. Les indices des liaisons permettent d'identifier les ports qu'elles relient.

$$Ass = (JPoint, L) \mid JPoint = \{port_{00}, ..., port_{nz}\} \; et \; L = \{\emptyset, l_{00,01}, ..., l_{n-1z,nz}\} \tag{4.20}$$

Afin de pouvoir instancier un tel modèle, il est donc nécessaire de mettre en œuvre un mécanisme synchronisant la représentation réelle d'un assemblage de composants d'une plateforme à composants et sa représentation abstraite. Ceci est réalisée à l'aide de deux transformations verticales [126], du modèle de l'application s'exécutant conforme au méta-modèle de la plateforme à composants cible vers un modèle conforme à notre méta-modèle d'assemblage de composants et vice versa. Cette transformation doit être automatique [126] et réalisée à l'exécution. Le mécanisme de transformation entre les modèles est propre à chaque plateforme, des correspondances doivent être établies entre les entités de chaque modèle. La table de correspondance 4.4 donne un exemple relatif à la plateforme LCA [87]. Cette synchronisation doit être réalisée à chaque fois qu'une modification intervient dans l'assemblage à adapter et lorsque le calcul de l'adaptation a été réalisé. Le tisseur prend donc en

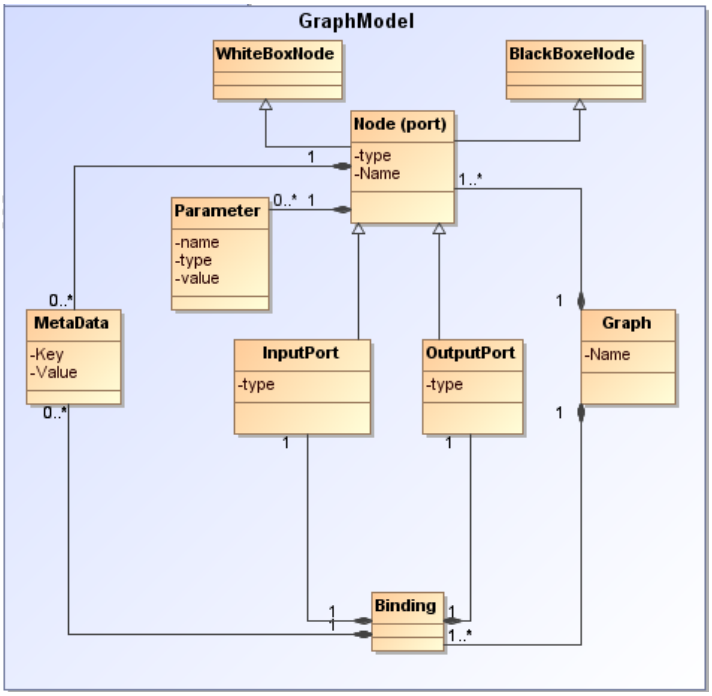


FIGURE 4.37 – Méta-modèle d'assemblage de composants.

entrée et produit des instances de ce méta-modèle.

TABLE 4.4 – Correspondance Assemblage de composants $\Longrightarrow$ LCA

Assemblage de composants	**LCA**
InputPort	Input/Method
OutputPort	Output/event
Node	Port
Binding	Link
WhiteBoxNode	Prédéfini
BlackBoxNode	Tous les autres
Parameter	Parameter
MetaData	Port.property

Les sections points de coupe de l'ensemble des instances de ce méta-modèle sont données en entrée au processus de pointcut matching du tisseur. Leurs sections greffon sont les entrées du processus d'advice factory. Ce processus prend en entrée des modèles de greffon, un modèle d'application et produit un ensemble de modèles d'application. Les greffons peuvent être vu comme des méta-modèles d'assemblage de composants comme nous pouvons le voir dans le tableau 4.5.

TABLE 4.5 – Correspondance Assemblage de composants $\Longrightarrow$ greffon d'AA

Assemblage de composants	**Greffon d'AA**
InputPort	InputPort
OutputPort	OutputPort
Node	PoincutVariable, local Entity, operator
Binding	RewriteInteractionRule, AddInteractionRule
WhiteBoxNode	Operator
BlackBoxNode	PoincutVariable, local Entity
Parameter	PointcutVariable.parameter
MetaData	PointcutVariable.property

4.4.3.2 Processus de tissage, transformation et composition

Un cycle de tissage repose donc sur trois transformations : (1) les entrées sont obtenues grâce au mécanisme de synchronisation et une transformation *application s'exécutant* $\longrightarrow$ *modèle abstrait de l'application* ; (2) le traitement consiste en une transformation horizontale de modèle endogène model-to-model [126] réalisée à l'exécution ; (3) la production de sorties consiste en une transformation verticale du mécanisme de synchronisation *modèle abstrait de l'application* $\longrightarrow$ *application s'exécutant*.

Ainsi la première phase d'un cycle de tissage fait partie du mécanisme de synchronisation *application s'exécutant* $\longleftrightarrow$ *modèle abstrait* de l'application. Dans cette phase, à chaque modification de l'assemblage s'exécutant, son modèle abstrait est mis à jour. Il s'agit d'une approche de synchronisation en mode *push* [96]. Le modèle peut alors être mis-à-jour de manière incrémentale, ce qui permet de faire évoluer en parallèle le modèle abstrait et l'application s'exécutant et évite d'introspecter l'ensemble de l'application s'exécutant pour identifier tous les changements (cf. 2.3.2.2). La seconde phase est une transformation horizontale de modèle endogène model-to-model [126] réalisée à l'exécution. Il s'agit d'une transformation automatique [126] d'un modèle assemblage de composants vers un modèle assemblage de composants. Cette transformation est paramétrée par les Aspects d'Assemblages. Sans paramètre ou si aucun AA n'est applicable alors la transformation n'introduit aucune modification de l'assemblage initial. Il s'agit donc d'une approche par transformation déclarative [126]. Enfin, lors de la troisième phase, lorsque le modèle abstrait est modifié, l'assemblage s'exécutant est mis à jour. Afin de ne modifier dans l'application que les parties qui ont changé, le modèle de l'application est comparé au modèle de l'application adaptée. Les mises à jour, afin de garantir la consistance de l'application et la mise œuvre de l'ensemble des reconfigurations, réalisent les règles de modification de l'architecture dans cet ordre :

1. retraits de liaisons ;
2. retraits de composants ;
3. ajouts de composants ;
4. ajouts de liaisons ;
5. modifications des propriétés.

La gestion du cycle de vie des composants est laissée à la charge soit de la projection spécifique vers une plateforme soit à la charge de la plateforme comme dans LCA. Puisqu'un cycle de tissage est

toujours réalisé sur l'assemblage initial tel que précédemment défini, les entités de la représentation abstraite de l'assemblage de composants sont identifiées comme étant ou non le résultat de l'adaptation. Pour cela, il convient de vérifier si les entités appartiennent à l'espace de nommage de l'AA et non pas à l'espace de nommage global auquel appartiennent les entités de l'assemblage initial. L'ensemble de ces processus est décrit en FIGURE 4.38.

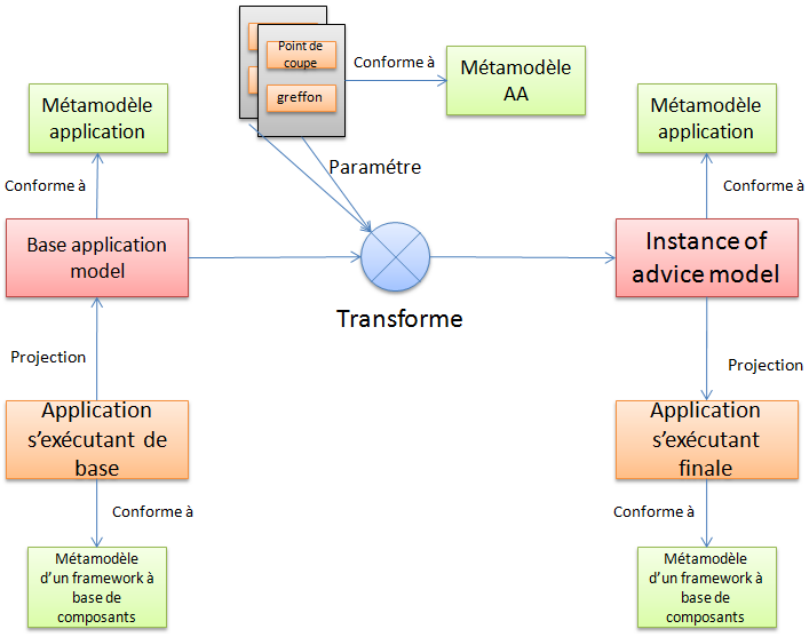


FIGURE 4.38 – Un cycle de tissage est une transformation endogène model-to-model.

Plus précisément, le processus de transformation *model-to-model* précédemment décrit se compose de deux transformations comme nous pouvons le voir en FIGURE 4.39. La première étape est une transformation endogène paramétrée par les AAs. Elle a pour but de transformer l'assemblage initial en plusieurs assemblages qui sont les instances de greffons. Cette transformation est réalisée dans une approche déclarative. La seconde étape est la composition endogène de ces instances de greffon afin de produire un unique assemblage de composants. Cette composition est dépendante des règles de fusion et des opérateurs de composition du moteur de composition. Il s'agit d'une transformation dans une approche *opérationnelle* [126]. Le moteur de composition travaille au niveau instances de greffon et non pas au niveau AA. En effet, au niveau type, toutes les interactions ne peuvent être identifiées puisque certaines sont dues aux duplications des AAs. A l'inverse, les règles données en paramètre au moteur de composition sont génériques et travaillent sur les types. Dans ces processus tous les assemblages produits sont conformes à notre méta-modèle d'assemblage de composants. Les deux transformations ont pour objectif de garantir la correction de l'application résultante des transformations. L'approche nous permet de définir à l'aide d'aspects des variantes d'architecture d'un système et de tisser les instances de ces variantes dans le modèle architectural [132].

Lors d'un tissage multi-cycles d'une cascade les transformations que nous venons de voir sont donc enchaînées les unes à la suite des autres comme cela est présenté en FIGURE 4.40. Dans ce cas, la synchronisation entre l'application modifiée et l'application s'exécutant ne se fait qu'une fois

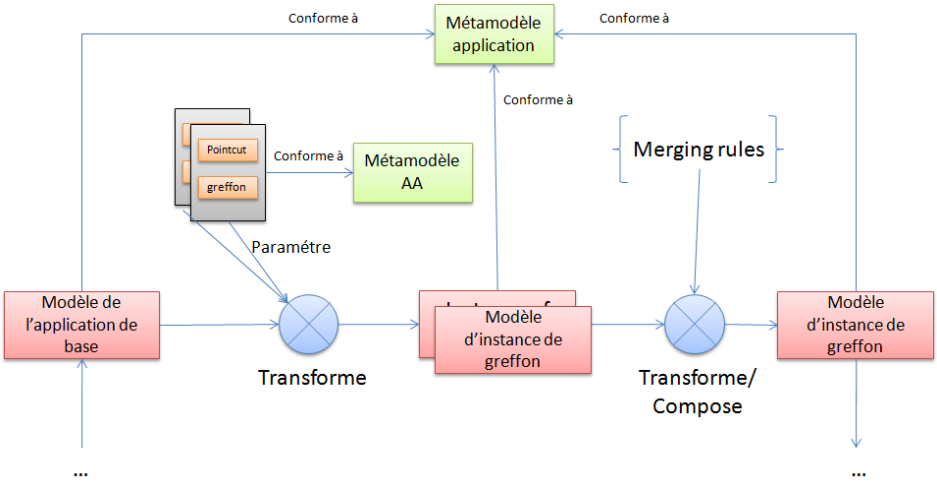


FIGURE 4.39 – Détail des transformations de modèles.

l'ensemble des transformations réalisées. Nous pouvons ainsi observer clairement qu'une adaptation est réalisée à l'aide de plusieurs processus de tissage. Pour cela, lors des trois phases du cycle de tissage, trois modèles d'assemblage sont finalement utilisés :

1. l'assemblage initial ;
2. l'assemblage courant ;
3. l'assemblage résultant de l'adaptation.

Le processus de synchronisation utilisant ces trois modèles peut donc être détaillé comme suit en cinq grandes étapes :

1. Mettre à jour l'application initiale. Pour cela, il faut filtrer les éléments de l'assemblage courant de ceux dus à une adaptation.
2. Mettre à jour l'image de l'application courante afin qu'elle soit à la disposition du mécanisme de synchronisation *modèle de l'application* $\longrightarrow$ *application s'exécutant*.
3. Si premier cycle de tissage alors transmettre l'application initiale aux chaînes de traitement des AAs du premier cycle. Pour les cycles suivants, transmettre le modèle résultat provenant de l'adaptation précédente. Le cycle en cours est déterminé par un compteur.
4. Après composition, mettre à jour le modèle représentant le résultat de l'application et déclencher le cycle suivant.
5. Une fois tous les cycles réalisés, faire une comparaison entre les modèles résultat et de l'application courante pour projeter dans l'application s'exécutant les différences.

Les cascades d'AAs respectent donc l'ensemble des contraintes que nous nous étions fixés en SECTION 4.1. L'ensemble de ces contraintes a pour conséquence de complexifier les processus mis en œuvre par le mécanisme d'adaptation et nécessairement sur les temps de réponse offerts. Nous allons maintenant étudier les temps de réponse des différents mécanismes que nous avons étudiés dans la présentation détaillée de notre tisseur.

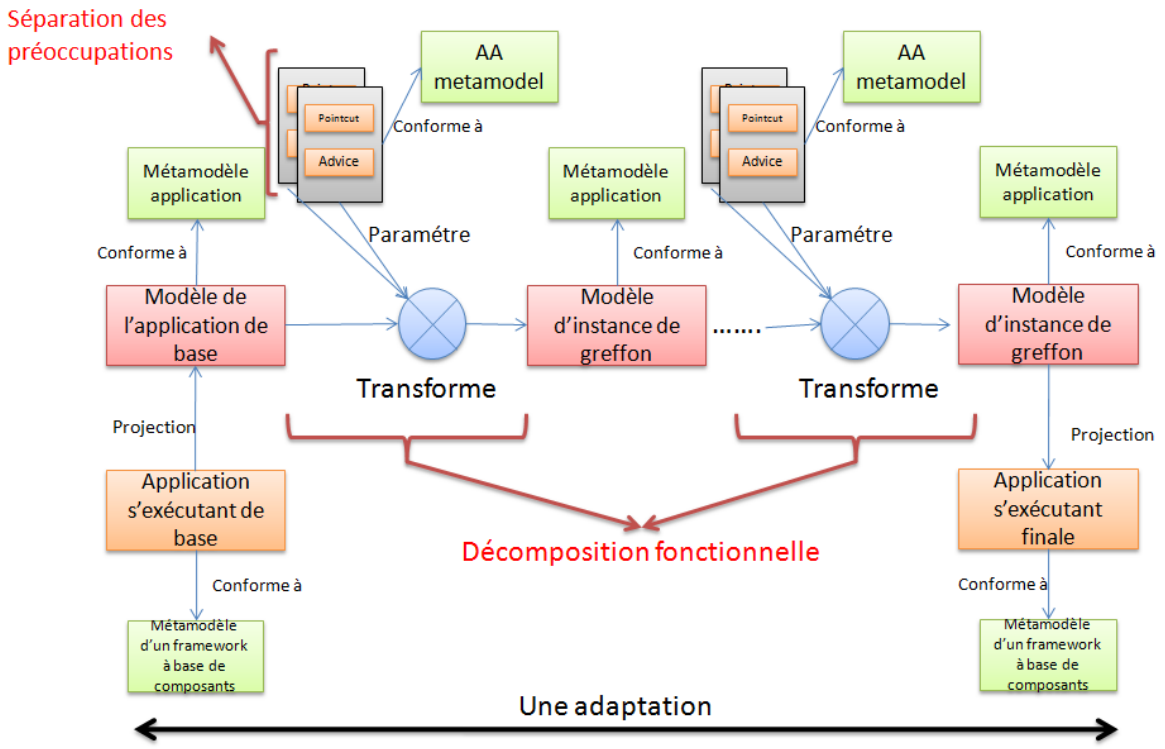


FIGURE 4.40 – Transformations mises en œuvre lors d'une adaptation à l'aide de cascades.

4.5 Propriétés temporelles

Nous avons vu qu'une des contraintes que doit respecter notre mécanisme d'adaptation est d'offrir des temps de réponse maîtrisés, finis et adaptés. Nous allons, dans cette section, étudier les temps de réponse proposés par les cascades d'AAs. Dans cette optique, après avoir identifié le coût de l'opération de tissage sur un cycle puis sur plusieurs cycles des opérations de pointcut matching et de composition, nous montrerons, dans un premier temps, que le coût de l'opération de tissage d'une cascade sur plusieurs cycles est borné par celui sur un cycle. Par conséquent, nous présenterons ensuite en détail les temps de réponse d'un cycle de tissage et des différents opérations nécessaires à sa réalisation.

4.5.1 Approche mono-cycle

Dans le pire des cas, l'opération de composition implique des règles qui interférent toutes les unes avec les autres et dont les interférences sont toutes différentes. Un tel cas nous permet de définir la borne supérieure du coût en temps de l'opération de composition. Ce coût est dépendant du nombre de règles à fusionner. Rappelons qu'un AA peut être écrit comme suit : $AA_i = \{pointcut_i, advice_i\}$ avec $advice_i = \{rule_{i0}, ..., rule_{ij}\}$. Nous pouvons donc définir le nombre de règles à composer comme :

$$nbRègles = \sum_{i=0}^{card(A_n)} card(advice_i) \tag{4.21}$$

Rappelons que l'opération de composition : $T(App_0, A_n) = App_1$ avec $An = \{AA_0, ..., AA_n\}$ l'ensemble des AAs sélectionnés et App_0 l'application de base, consiste à composer entre elles, les règles des instances de greffon obtenues à partir de An ainsi que les règles de la représentation abstraite de l'assemblage de base. Les règles de l'assemblage de base sont considérées comme sans interférences. Le coût en temps, dans le pire des cas, de l'opération de composition dans l'approche mono-cycle peut être exprimé comme suit :

$$cT = [(2^{nbRègles} - (nbRègles + 1)) \times card(App_0)] \times (coût\ de\ la\ fusion\ de\ deux\ règles) \quad (4.22)$$

En effet, en considérant que l'opération de fusion est symétrique et que toutes les règles sont en conflit (et que toutes les interférences sont différentes les uns des autres), le nombre d'opération de fusion de deux règles à réaliser est de : $(2^{nbègles} - (nbRègles + 1))$.

Il en va de même pour le processus de reconnaissance des points de coupe. Rappelons que ce processus a pour objectif d'identifier des ensembles de points de jonctions sur lesquels les greffons devront être appliqués. Chaque règle produit un ensemble de point de jonction. Par la suite, le processus calcul par défaut l'ensemble des combinaisons de points de jonctions. Une combinaison de points de jonction est un n-uplet dont la cardinalité est égale au nombre de règles dans le point de coupe. Chaque élément de ce n-uplet provient d'une règle différente. Ces traitements étant réalisés indépendemment pour chaque AA, le coût en temps, dans le pire des cas, de l'ensemble de processus de pointcut matching est le pire des coûts de tous les aspects. Il dépend du nombre de combinaisons qui doivent être calculées, à savoir :

$$cJP = nbJPoint^{card(pointcut)} \quad (4.23)$$

4.5.2 Approche multi-cycles

Dans l'approche multi-cycles, le temps passé afin de gérer l'enchaînement des cycles de tissage ainsi que l'historique des assemblages de base est minimal. Comme nous pouvons le voir en FIGURE 4.41, ce temps augmente en fonction au nombre de cycles impliqués dans la cascade, l'historique des assemblages de base augmentant avec lui. Cette FIGURE 4.41 présente la durée du processus de tissage en fonction du nombre de points de jonction du processus de tissage sans les étapes de composition et de fusion dans les cycles 1, 2 et 3 d'une cascade. Dans chacun des cycles, les mêmes AAs ont été déployés, de cette manière nous pouvons voir clairement comment évolue la durée de gestion des cascades au fil des cycles de tissage.

Comme pour l'approche par tissage mono-cycle, dans le pire des cas, l'opération de composition implique dans chaque cycle, des règles qui interférent toutes les unes avec les autres, avec des interférences toutes différentes. Le nombre de règles à composer dans un cycle est donc de :

$$nbRègles_i = \sum_{j=0}^{card(A_i)} card(advice_i) \quad (4.24)$$

Rappelons que l'opération de composition dans l'approche multi-cycles : $App_m = T_m(A_m, T_{m-1}(A_{m-1}, ..., T_0(A_0, App_0)))$ avec A_m l'ensemble d'AAs destiné au cycle m, consiste à composer entre elles les règles des instances de greffon obtenues à partir de A_m ainsi que les règles de la

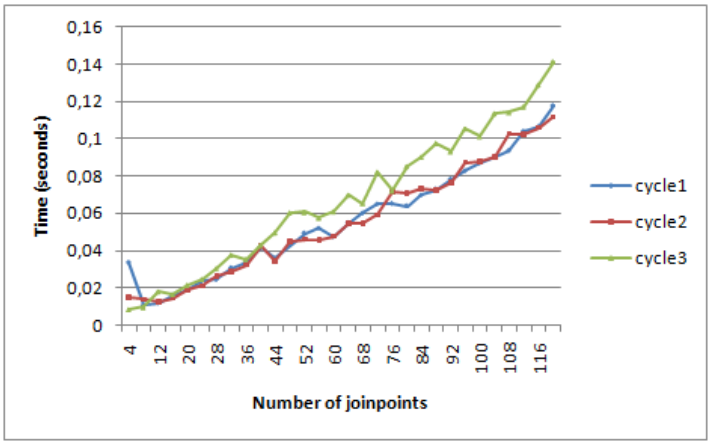


FIGURE 4.41 – Cascades d'AAs.

représentation abstraite de l'assemblage de base de ce même cycle. Le coût en temps, dans le pire des cas, de l'opération de composition dans l'approche multi-cycles peut être exprimé comme suit :

$$cT^m = [\sum_{i=0}^{m}(2^{nbRègles_i} - (nbRègles_i + 1)) \times card(App_i)] \times (coût\ de\ la\ fusion\ de\ deux\ régles) \tag{4.25}$$

En effet, en considérant que l'opération de fusion est symétrique et que toutes les règles sont en interférence (et que tous les interférences sont différents les uns des autres), le nombre d'opération de composition de deux règles à réaliser est la somme des nombres d'opérations de composition de chaque cycle : $\sum_{i=0}^{m}(2^{nbRègles_i} - (nbRègles_i + 1))$.

Pour la reconnaissance des points de coupe, nous avons vu précédemment que le coût de cette opération dépend du nombre de combinaisons qui doivent être calculé, dans l'approche multi-cycles, ce nombre est la somme des combinaisons qui doivent être calculées lors de chaque cycle.

$$\sum_{j=1}^{m} nbJPoint^{card(pointcut_j)} \tag{4.26}$$

4.5.3 Synthèse

Nous pouvons donc voir que pour mettre en œuvre une même fonctionnalité dans une application, c'est à dire en décomposant des AAs en cascades d'AAs sans ajouter de fonctionnalité particulière, en fonction de l'approche choisie, le coût de l'opération de composition peut varier. Ainsi si nous considérons que $A_n = A_m \cup ... \cup A_0$, nous pouvons conclure que dans le pire des cas $cT^m \leq cT$ puisque :

$$[\sum_{i=0}^{m}(2^{nbRègles_i} - (nbRègles_i + 1)) \times card(App_i)] \leq [(2^{nbRègles} - (nbRègles + 1)) \times card(App_0)] \tag{4.27}$$

Il en va de même pour la reconnaissance des points de coupe. Le coût de l'approche multi-cycles est borné dans le pire des cas par l'approche mono-cycle :

$$\sum_{j=1}^{m} nbJPoint^{card(pointcut_j)} \leq \prod_{i=1}^{m} nbJPoint^{card(pointcut_i)} \tag{4.28}$$

Nous pouvons donc conclure que les temps de réponse des adaptations réalisées à l'aide de cascades d'AAs sont bornés par les temps de réponse des adaptations réalisées à l'aide d'AAs. Ainsi décomposer un AA en une cascade d'AAs et ainsi réaliser un tissage multi-cycles et ainsi augmenter la variabilité du mécanisme d'adaptation n'est pas un facteur limitant par rapport aux temps de réponse.

4.5.4 Étude approfondie sur un cycle de tissage

Nous allons maintenant étudier les différents temps de réponse des étapes réalisées lors d'un cycle de tissage. Pour chacun de ces éléments, nous donnerons :

1. un rappel de l'objectif du mécanisme ;
2. la complexité en temps de l'algorithme ;
3. un modèle mathématique permettant de calculer la durée de mise en œuvre du processus ;
4. des évaluations de performances. Différentes expérimentations ont étés réalisées sur un ordinateur portable standard (Athlon X2, 1.6GHz, 512 Mo RAM).

Les évaluations de performances sont réalisées sur le même scénario fil rouge que nous avons utilisé en SECTION 4.3.3.

4.5.4.1 Pointcut Matching

Pour rappel, le pointcut matching et le processus qui a pour but de déterminer dans l'assemblage de base tous les endroits où les modifications décrites dans les greffons des AAs peuvent être appliquées. Nous avons vu que la complexité algorithmique de ce mécanisme est de :

$$O((j \times card(JPoint) \times O(ml)) = O(j \times card(JPoint) \times m \times l)$$

Nous pouvons déduire de cette complexité le modèle mathématique de la durée du processus de poincut matching présenté en FIGURE 4.42.

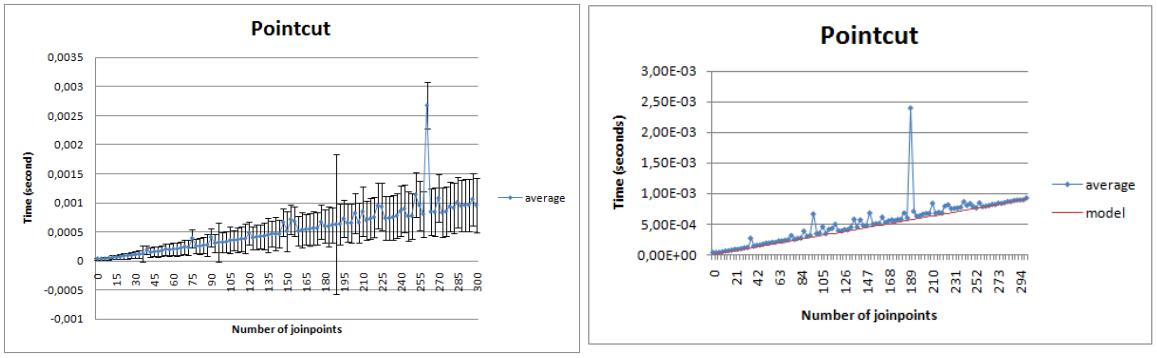
D : durée du processus de pointcut matching
$a1$; $a2$: paramètres du modèle
c : nombre de points de jonction
i : nombre d'AAs
j : nombre de règles dans la section point de coupe de l'AA

$$D = a1 \times \sum_{k=1}^{i} (j_k.c) + a2$$

FIGURE 4.42 – Durée du processus de pointcut matching

Les expérimentations du pointcut matching ont mis en jeu le point de coupe précédemment présenté composé de trois règles, ainsi qu'un ensemble de points de jonction que nous avons fait varier

de 0 à 300. Plusieurs séries d'expérimentations ont étés réalisées, la courbe présentée en FIGURE 4.43a est une moyenne de ces séries et montre l'écart type entre chacune des valeurs de ces séries. La Figure 4.43b permet de comparer ces expérimentations à notre modèle mathématique. Nous avons extrait des expérimentations les valeurs des paramètres du modèle $a1 = 10^{-6}$ et $a2 = 1,4 \times 10^{-6}$. Ces paramètres du modèle sont des valeurs propres au système sur lequel ont étés réalisées les évaluations. Nous voyons donc que le processus de pointcut matching est rapide et peu coûteux en temps. Bien entendu ce coût est fortement dépendant du type de filtre utilisé.



(a) Moyenne des temps de réponse et écarts type.

(b) Moyenne des temps de réponse et le modèle mathématique associé.

FIGURE 4.43 – Temps de réponse du processus de pointcut matching.

4.5.4.2 Combinaisons de points de jonction

Le processus de combinaison des points de jonction, aussi appelé (JoinPoint Combination) a pour objectif de combiner les points de jonction vérifiant le pointcut matching selon une politique associée à l'AA dans l'optique de définir *où* et *comment* doivent être dupliqués les AAs. Une combinaison de points de jonction est une liste de points de jonction ayant vérifiés le pointcut matching. La cardinalité d'une combinaison est égale au nombre de règles se trouvant dans le pointcut de l'AA, chaque point de jonction dans une combinaison ayant vérifié une règle de point de coupe différente. Nous avons vu que divers algorithmes de combinaisons peuvent être implémentés. L'algorithme choisi consiste à calculer toutes les combinaisons possibles (ALGORITHME 2) dont la complexité algorithmique est de :

$$O(card(JPoint)^j).$$

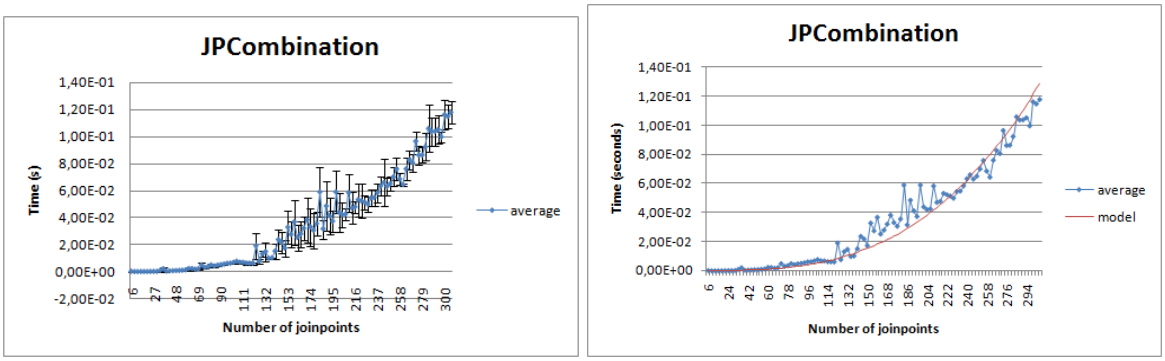
Nous pouvons déduire de cette complexité le modèle mathématique de la durée du processus de JoinPoint Combination présenté en FIGURE 4.44.

Les expérimentations ont mis en jeu le point de coupe précédemment présenté composé de trois règles, ainsi qu'un ensemble de points de jonction que nous avons fait varier de 0 à 300. Plusieurs séries d'expérimentations ont étés réalisées, la courbe présentée en FIGURE 4.45a est une moyenne de ces séries et montre l'écart type entre chacune de leurs valeurs. La FIGURE 4.45b permet de comparer ces valeurs expérimentales à notre modèle mathématique. Nous avons extrait des expérimentations les valeurs des paramètres du modèle $a1 = 0.45 \times 10^{-6}$ et $a2 = 1 \times 10^{-6}$. Nous voyons donc que la durée du processus de combinaison des points de jonction peut être élevée en fonction de l'algorithme choisi.

C : durée du processus de combinaison des points de jonction
$a1$; $a2$: paramètres du modèle
$JPoint$: ensemble des points de jonctions
i : nombre d'AAs
j : nombre de règles présente dans la section point de coupe de l'AA

$$C = a1 \times \sum_{k=1}^{i} (card(JPoint)^{j_k}) + a2$$

FIGURE 4.44 – Durée du processus de combinaison des points de jonction.



(a) Moyenne des temps de réponse et écarts type.

(b) Moyenne des temps de réponse et le modèle mathématique associé.

FIGURE 4.45 – Temps de réponse du mécanisme de combinaison des points de jonction.

4.5.4.3 Fabrique de greffons

L'objectif de la fabrique de greffon est de construire, à partir de la liste des combinaisons de points de jonction, des instances de greffon. Elle permet de relier les greffons qui sont des représentations abstraites d'assemblages de composants à l'assemblage de base et ainsi de réaliser autant d'instances de greffon qu'il y a de combinaisons. Nous avons vu que la complexité algorithmique de ce mécanisme est la suivante :

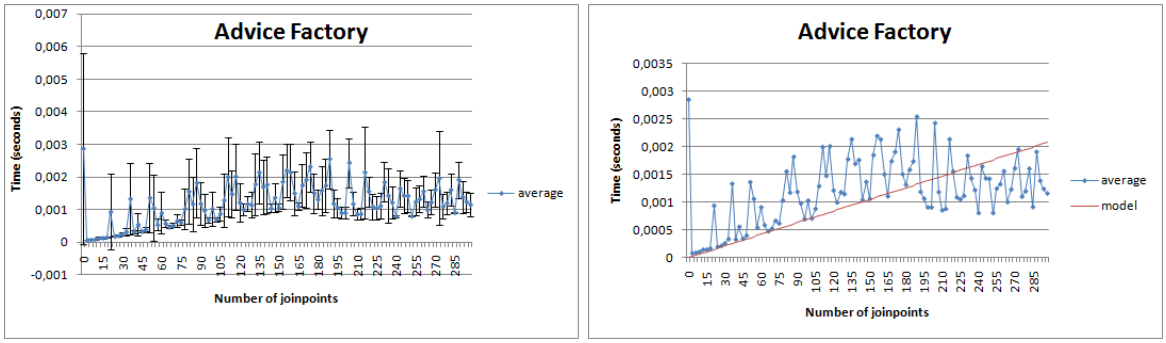
$$O(k \times w)$$

Nous pouvons déduire de cette complexité le modèle mathématique permettant de calculer la durée du processus de fabrication d'instances de greffon présenté en FIGURE 4.46.

A : durée de la génération d'instances de greffon
k : nombre de combinaisons
w : nombre de règles
i : nombre d'AAs
$a1$;$a2$: paramètres du modèle

$$A = a1 \times \sum_{p=1}^{i} (kw_p) + a2$$

FIGURE 4.46 – Duration of instance of advice generation



(a) Moyenne des temps de réponse et écarts types

(b) Moyenne des temps de réponse et le modèle mathématique associé

FIGURE 4.47 – Temps de réponse de la fabrique de greffons.

Les expérimentations ont mis en jeu le point de coupe précédemment présenté composé de trois règles, et donc des combinaisons composées de trois points de jonction. Nous avons fait varier le nombre de points de jonction de 0 à 300. Plusieurs séries d'expérimentations ont étés réalisées, la courbe présentée en FIGURE 4.47a est une moyenne de ces séries et montre l'écart type entre chacune des valeurs de ces séries. La FIGURE 4.47b permet de comparer ces expérimentations à notre modèle mathématique. Nous avons extrait des expérimentations les valeurs des paramètres du modèle $a1 = 3 \times 10^{-6}$ et $a2 = 1 \times 10^{-6}$. Ce processus est donc très peu coûteux en temps.

Dans l'ensemble, parmi tous les processus travaillant sur les AA, le calcul des combinaisons de points de jonction est le plus long. Cependant, en fonction de l'algorithme associé à l'AA, celui-ci peut-être fortement amélioré. Nous pouvons conclure que les traitements réalisés sur les AAs sont peu coûteux en temps.

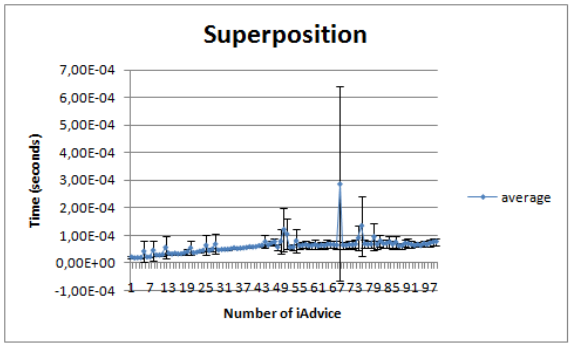
4.5.4.4 Superposition

La superposition d'assemblages de composants est une expression qui construit systématiquement un unique assemblage résultant à partir de plusieurs assemblages de composants intermédiaires. Nous avons vu que la complexité de cet algorithme est de : $O(y \times max(Card(iAdvise_i)) \times card(G_0))$. Il consiste à itérer sur toutes les règles de toutes les instances de greffon est de les ajouter dans G_0 en vérifiant préalablement si elle n'y sont pas déjà. Nous pouvons déduire de cette complexité le modèle mathématique de la durée du processus de superposition présenté en FIGURE 4.48.

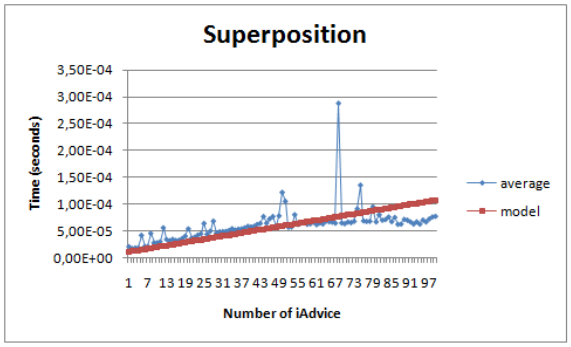
S : durée de la superposition d'instances de greffon
y : nombre d'instances de greffon
w : nombre de règle dans l'advice de l'AA
$g0$: nombre de règles dans l'assemblage initial
$a1$;$a2$: paramètres du modèle

$$S = a1 \times \sum_{i=1}^{y} (w_i.g_0) + a2$$

FIGURE 4.48 – Durée de la superposition d'instances de greffon.



(a) Moyenne des temps de réponse et écarts types.



(b) Moyenne des temps de réponse et le modèle mathématique associé.

FIGURE 4.49 – Temps de réponse du mécanisme de superposition.

Les expérimentations ont mis en jeu un ensemble d'instances de greffon dont nous avons fait varier la cardinalité de 0 à 100. Ces instances sont composées de sept règles. Plusieurs séries d'expérimentations ont étés réalisées, la courbe présentée en FIGURE 4.49a est une moyenne de ces séries et montre l'écart type entre chacune des valeurs de ces séries. La FIGURE 4.49b permet de comparer ces expérimentations à notre modèle mathématique. Nous avons extrait des expérimentations les valeurs des paramètres du modèle $a1 = 3 \times 10^{-6}$ et $a2 = 1 \times 10^{-6}$. Ce processus est donc peu coûteux en temps.

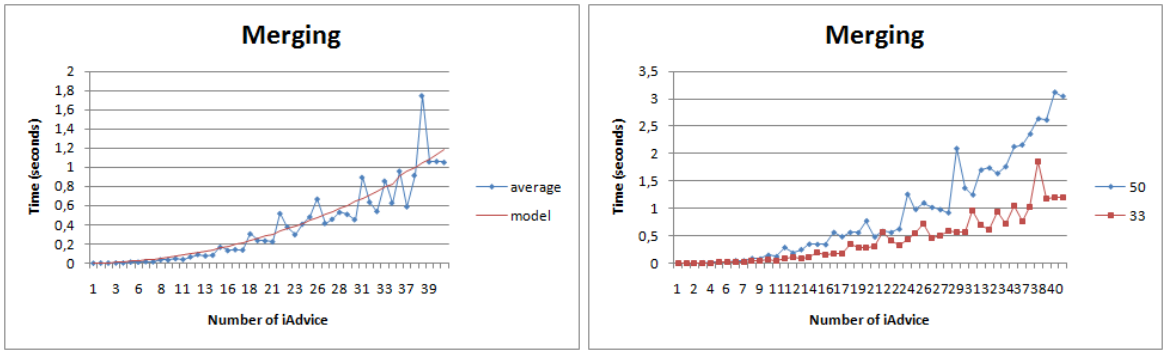
4.5.4.5 Fusion

Le mécanisme de résolution des conflits a pour but de gérer les interactions qui peuvent intervenir lorsque plusieurs instances de greffon sont tissées sur un même point de jonction (en anglais, shared joinpoint) tout en conservant la consistance de l'application et en garantissant la propriété de symétrie de l'opération de tissage. Les expérimentations que nous avons réalisées sont basées sur l'implémentation présenté en [64] du mécanisme de gestion des interférences ente AAs. Cette implémentation est basée sur ISL4WComp. Dans cette implémentation, nous pouvons définir M comme proportionnel à la plus petite hauteur des arbres à fusionner, de telle manière que : $M = k_0.min(h_0;h_i)$ avec k_0 un paramètre du modèle dépendant du système sous-jacent [64]. La FIGURE 4.50 présente le modèle mathématique permettant de calculer la durée de l'opération de gestion des interférences entre AAs.

F : durée de la fusion d'instances de greffon
g_o : nombre de règles dans l'assemblage initial
y : nombre d'instance de greffon
w_i : nombre de règle dans le greffon
$a1$: paramètre du modèle
p_i : probabilité de fusion
M : coût de la fusion de deux règles

$$F = a1.g_0 \times \sum_{i=1}^{y} w_i.p_i.M$$

FIGURE 4.50 – Durée de la fusion d'instances de greffon.



(a) Moyenne des temps de réponse et le modèle mathématique associé.

(b) Moyenne des temps de réponse avec p=33% et 50%.

FIGURE 4.51 – Temps de réponse du mécanisme de résolution des conflits.

Les expérimentations ont mis en jeu un ensemble d'instances de greffons dont nous avons fait varier la taille de 0 à 50. La courbe présentée en FIGURE 4.51a, basée sur nos résultats expérimentaux, montre l'évolution du temps requis pour réaliser la gestion des interférences entre des AAs dont la probabilité d'avoir une interférence entre deux aspects est de 0.33. Cette figure présente également notre modèle mathématique. Nous pouvons extraire de ces résultats le paramètre du modèle suivant : $a1 = 1 \times 10^{-6}$. La courbe présentée en FIGURE 4.51b présente également des résultats expérimentaux. Les courbes représentent l'évolution du temps requis pour réaliser la gestion des interférences entre des AAs dont les probabilités d'avoir une interférence entre deux aspects sont de 33% et de 50%. Ces évaluations illustrent bien l'importance du mécanisme de gestion des interférences entre AAs dans le processus de tissage. En effet, la durée de ce processus représente environ 85% de la durée total de l'opération de tissage. Nous pouvons également observer que la probabilité d'avoir un conflit et donc le nombre de conflits mis en jeu joue un rôle majeur pour les performances du tissage.

4.5.4.6 Synthèse

L'ensemble de ces résultats expérimentaux et des modèles définis nous permet de décrire l'opération de tissage comme proposant des temps de réponse raisonnables et maîtrisés. Ainsi, lorsque nous calculons toutes les combinaisons possibles de points de jonction, nous pouvons calculer la durée de l'opération de tissage à l'aide du modèle mathématique suivant présenté en FIGURE 4.52.

L'essentiel du temps de tissage se déroulant dans la phase de fusion lorsque celle-ci intervient. La durée d'un cycle de tissage est égale à la somme des durées des différents traitements intervenant dans le processus de tissage. Pendant cette période, le tisseur ne considère plus les nouvelles perturbations de son environnement, c'est-à-dire qu'il ne considère plus les nouveaux changements intervenant dans son environnement ou la sélection/désélection d'AAs par l'utilisateur. Ainsi, en fonction de la durée W, la fréquence des perturbations tolérées est de :

$$F = 1/W$$

Grâce au modèle mathématique précédemment défini, nous pouvons produire la FIGURE 4.53. Cette dernière représente la durée d'un cycle de tissage en fonction du nombre de points de jonction présent dans l'assemblage de base. Nous considérons dans cette courbe que tous les points de jonction vérifient le pointcut matching et que les combinaisons sont générées en fonction de l'affinité des

W : durée du processus de tissage
g_o : nombre de règle dans l'assemblage initial
w_i : nombre de règle dans le greffon de AA_i
$a1, a2$: paramètres du modèle
p_i : probabilité de fusion
M : coût de la fusion de deux règles
c : nombre de points de jonction
i : nombre d'AAs
j_i : nombre de règles dans les points de coupe de AA_i
p_{kj} : probabilité qu'un point de jonction satisfasse une règle de point de coupe

$$W = a1.g_0 \times \sum_{k=1}^{i}[(j_k.c) + \sum_{z=1}^{j_k}(p_{kj}.c)^{j_k} + NBComb.w_k + \sum_{l=1}^{NBComb} w_k.p_k.M] + a2$$

$$NBComb = \prod_{r=1}^{j_k}(2^{(p_{km}.c)-1})$$

FIGURE 4.52 – Durée du processus de tissage.

méta-données associées (ici le nom) aux points de jonction. Dans cet exemple, le mécanisme de fusion est requis dans 33% des cas et deux AAs sont sélectionnés ; ce sont les AAs présentés en FIGURE 4.23.

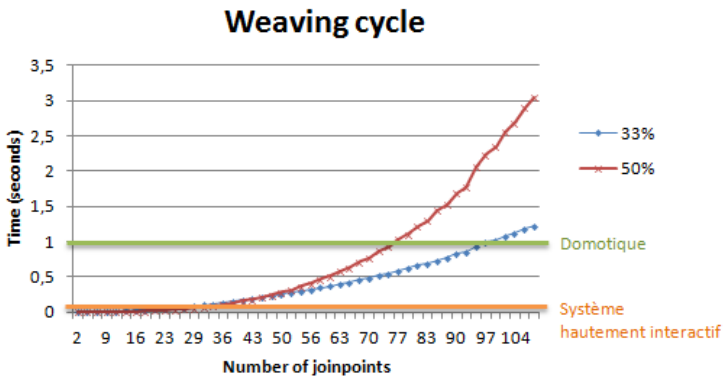


FIGURE 4.53 – Durée d'un cycle de tissage.

Nous avons vu en SECTION 4.2.4 qu'il est considéré que la « latence utilisateur » est, dans le pire des cas, de 100ms. F. Bérard dans [138] propose alors que la latence des systèmes hautement interactifs doit être moitié moins importante que celle de l'utilisateur : 50ms. En dessus de cette borne, nous sommes capable de composer environ 30 points de jonction les uns avec les autres. Toutefois, dans le cadre de l'informatique ambiante, de telles latences ne sont pas nécessairement requises. Ainsi, par exemple dans le cadre de la domotique, une latence d'une seconde est tolérée. En dessus de cette borne, nous sommes capables de composer environ 100 points de jonction les uns avec les autres.

Dans le cadre du projet ANR Continuum, une mise en œuvre réelle de l'approche a été réalisée. Ce scénario prend place dans le cadre du travail de fontainier. Une des tâches d'un fontainier est de fermer des vannes dans un réseau de canalisation d'eau dans le but de réaliser des opérations de maintenance sur le réseau. Le fontainier lorsqu'il part en mission est équipé d'un ensemble d'équipe-

ments portables et dans sa voiture (GPS, Radar, Map, Camera, boussole ...). Les vannes peuvent elles aussi être équipées de dispositifs (capteurs d'humidité, RFid ...). Dans le cadre de ce scénario 18 AAs ont étés écrits pour environ 25 règles et entre 7 à 10 dispositifs sont utilisés ainsi que 7 composants logiciels sur étagère pour l'interface utilisateur. Ainsi, le nombre d'instances d'advice varie entre 20 et 30 en fonction des dispositifs présents. Les nombre de conflits identifiés est compris entre 5 et 10 soit environ dans 35% des cas. Les temps de réponse observés et proposés sont inférieurs à 50ms.

Nous avons décrit ici les temps de préparation de l'adaptation qui doit être appliqué. Pendant ce temps, l'application est toujours en exécution. La latence entre l'occurence d'un changement et la réaction à ce changement, dépend de la durée de calcul de l'adaptation mais aussi de l'application de l'adaptation et de l'observation du changement. La durée de l'application de l'adaptation dépend de la plateforme d'exécution sous-jacente. Dans le cadre de WComp, le temps de création d'un composant simple comme le composant *if* est de l'ordre de 0.048ms. Ces temps de réponse restent difficiles à comparer à d'autres travaux en raison du peu d'études de ce type existant dans la littérature. Cependant, les valeurs proposées semblent moyennes puisque, dans une étude réalisée sur openCOM [66], il est mis en avant un temps d'instanciation de composant vide de l'ordre de 0,47μs tandis que dans une étude sur des composants distribués .Net [104] évoque des temps de l'ordre de 5ms. La création d'une liaison est de l'ordre de 0.43ms. Dans le cadre de notre scénario fil rouge, en fonction des dispositifs présents, entre 10 et 20 composants sont instanciés et une vingtaine de liaisons sont créées ou supprimées. Nous pouvons considérer que ce processus est d'environ 9ms. Le temps de mise en œuvre de l'adaptation, et donc la durée de la modification de l'application, est donc faible et prend une faible place dans la latence évoquée précédemment. A une plus grande échelle, pour 200 composants, les temps observés sont de 9.6ms. Sachant que le système peut, par exemple, s'appliquer au niveau de plusieurs conteneurs locaux capables de communiquer entre eux, l'architecture globale du système peut donc prendre diverses formes. Dans une approche ou de nombreux conteneurs sont utilisés et collaborent, 200 composants peuvent être considérés comme une borne haute du nombre de composants manipulés par une adaptation dans un conteneur. Par contre, lorsqu'un unique conteneur est utilisé le nombre de composant peut alors facilement dépasser cette borne. Les mesures sur l'instanciation de 10 000 composants, montre un temps de réponse raisonnable de l'ordre de la seconde.

A l'inverse, le temps de monitoring peut être important car il met bien souvent en jeu des communications réseau. Considérons un des cas de déclenchement d'une adaptation qui est l'apparition d'un composant. Comme nous l'avons vu, nous souhaitons que l'apparition d'un composant puisse être liée à l'entrée dans le réseau d'un dispositif. Dans WComp, des composants proxies vers des services pour dispositifs UPnP [1] peuvent être générés dynamiquement puis instanciés à l'aide des mécanismes de découverte dynamique de UPnP. Nous avons mesuré la découverte d'un device UPnP composé d'un service utilisant la pile C♯ dans un réseau local en filaire à environ 190ms. Ce chiffre augmente fortement dans un réseau Wifi, avec peu de traffic, la découverte d'un même device est mesurée à 345ms. A ce temps de découverte, il faut ajouter, lors de la première apparition de ce service, le temps de générer le composant proxy permettant de communiquer avec lui qui est environ de 260ms. L'observation des changements de ce cas peut alors être estimée dans de bonnes conditions à 190ms ou 440ms pour un nouveau service. Cette durée prend une part non-négligeable voir la plus importante dans la latence entre le changement et la réaction à ce changement et est fortement dépendante des conditions d'utilisation du réseau. Des améliorations restent à apporter dans ce domaine.

1. Universal Plug and Play

Troisième partie

Validation et application

CHAPITRE 5

Mise en œuvre de l'architecture sur 4 niveaux et du mécanisme d'adaptation

« C'est l'imprévisible qui crée l'événement. »

Georges Braque.

Sommaire

DANS ce chapitre nous présentons l'implémentation des concepts que nous avons présentés précédemment. A partir d'un scénario, que nous appliquerons au cadre du bâtiment, nous proposerons une mise en œuvre de l'architecture sur 4-niveaux. Cette architecture, qui a été utilisée dans le cadre du projet Continuum, reposera sur les implémentations suivantes : une infrastructure logicielle de services pour dispositifs, la plateforme d'exécution WComp, un designer de cascades d'AAs et un gestionnaire de contexte qui a été développé dans le cadre du projet Continuum.

5.1 Scénario

Le scénario proposé se déroule dans le cadre d'un bâtiment intelligent. Seule une partie des services offerts par le système ambiant est présentée dans ce scénario :

Comme tous les jours, Bob se rend au bureau. Ce dernier se trouve au siège social de son entreprise. Au centre de ce bâtiment se trouve un dôme en verre. Les fenêtres de ce dôme se trouvent dans le hall d'exposition dans lequel est déployé un ensemble de matériel informatique. Les fenêtres du dôme s'ouvrent lorsque la température à l'intérieur du bâtiment est trop élevée et se ferment lorsqu'il pleut. Dans le hall, des capteurs permettent de détecter des inondations afin de protéger le matériel. Ainsi, en cas de dysfonctionnement des fenêtres lors d'une pluie, une alerte est directement envoyée sur l'ordinateur du gardien. Au cas où le gardien serait absent, une alerte est également transmise aux personnes se trouvant dans le bâtiment afin que l'une d'entre elle ferme manuellement les fenêtres.

Aujourd'hui, des réparateurs viennent s'occuper du dôme, une vitre ayant été endommagée. Pour ne pas être pris au dépourvu par le fonctionnement des fenêtres, ils en prennent le contrôle. Alors qu'ils sont en pleine opération sur le dôme, un petit orage se déclenche. Les techniciens, non conscients de la présence du matériel se trouvant dans le hall, sont également avertis du risque.

Ce scénario nous permet de mettre en avant les propriétés suivantes :

- **Temps de réponse** : ce besoin apparaît à plusieurs moments dans le scénario. Il apparaît dans un premier temps pour le message d'alerte destiné au gardien puis dans les alertes destinées aux personnes se trouvant dans le bâtiment.

- **Variabilité** : la variabilité s'exprime à la fois par la diversité des dispositifs présents mais aussi par les changements intervenant dans l'ensemble des dispositifs disponibles pour les messages d'alerte. Ces changements peuvent être dus à des personnes qui entrent ou sortent, ou encore aux possibles dysfonctionnements de dispositifs. Enfin, elle s'exprime par l'ajout de nouveaux comportements par les techniciens.

- **Situations** : le besoin de faire varier l'ensemble des comportements déployés apparaît avec la mise en place dans ce scénario de deux situations qui sont les états avant et pendant l'averse. Le besoin de déployer de nouveaux comportements en cas de pluie pour avertir les personnes se trouvant dans le bâtiment apparaît bien dans ce scénario. En effet, on ne peut pas laisser déployer ce comportement et laisser le système s'adapter, sans que cela ne soit nécessaire, à chaque fois qu'une personne entre ou sort du bâtiment. Ceci est d'autant plus vrai que le nombre de personnes assistant à une démonstration dans le hall peut être élevé. Réaliser ces adaptations risque de ralentir la mise en place d'autres comportements décrits ou non dans ce scénario.

- **Imprévisibilité et extensibilité** : l'imprévisibilité vient tout d'abord de la disponibilité des dispositifs se trouvant dans le bâtiment. D'autre part, elle provient également de l'ajout, non anticipé, d'un nouveau comportement par les techniciens. Cette extension fait émerger un nouveau comportement du système ambiant lorsqu'elle est fusionnée avec les comportements existants.

- **Utilisation de l'adaptation compositionnelle pour faire de l'adaptation paramétrée** : les comportements d'alerte déployés en cas de pluie, prennent la forme d'adaptations compositionnelles puisque l'ensemble des dispositifs prévus ne peut pas être anticipé. L'objectif de cette adaptation est de mettre en place dans l'application un mécanisme d'adaptation paramétrée. A l'inverse, le comportement concernant l'alerte lancée au gardien, s'appuie sur un dispositif fixe et connu à l'avance. Il est donc toujours déployé et permet de réaliser une adaptation paramétrée. L'adaptation dans ce cas consiste, sur changement de valeur et dépassement d'un seuil, à modifier un paramètre qui permet d'envoyer un message au gardien sans modification structurelle.

- **Multi-dynamique** : les deux types d'alertes mettent en avant le besoin de disposer d'une approche multi-dynamiques. Pendant que le message d'alerte est envoyé au gardien, le système identifie le besoin de changer de situation et de déployer de nouveaux comportements. Ces derniers sont ensuite évalués afin d'adapter l'application.

A partir de ce scénario, nous allons dans un premier temps présenter notre mise en œuvre de l'architecture sur 4-niveaux. Il est important de noter que les approches présentées sont données à titre d'exemple et que d'autres approches auraient pu être utilisées pour réaliser les différents niveaux de l'architecture.

5.2 Mise en œuvre de l'architecture

L'architecture globale du système ambiant mis en place pour ce scénario est définie comme représenté sur la FIGURE 5.1. Elle reposera en partie sur les travaux réalisés dans le cadre du projet ANR Continuum. Ce projet traite du problème de la continuité de service en intelligence ambiante [1]. Nous allons détailler dans les sections suivantes les différents niveaux de cette architecture. Dans un

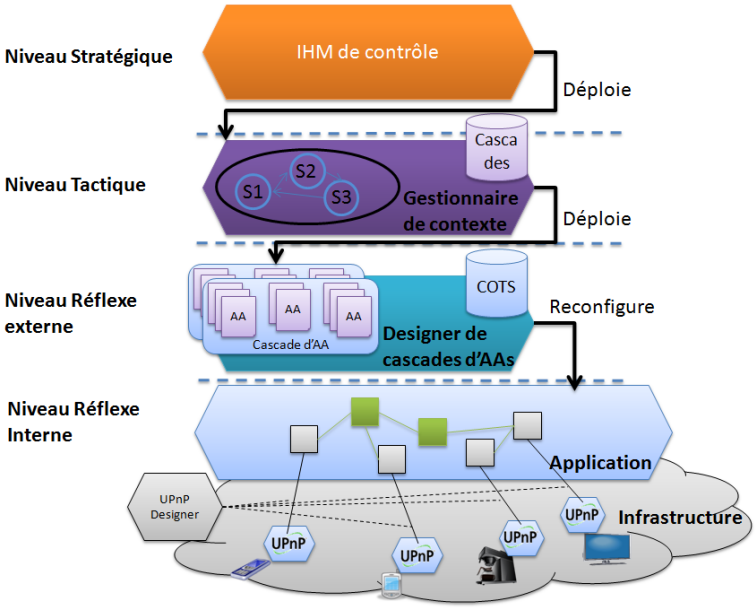


FIGURE 5.1 – Architecture générale.

premier temps nous présenterons l'approche orientée services pour dispositifs que nous avons retenue pour mettre en place l'infrastructure logicielle du système. Ensuite, nous étudierons la plateforme d'exécution à base de composants que nous avons utilisée et qui nous permet de faire des orchestrations des services de l'infrastructure. Cette plateforme nous permettra de mettre en place notre niveau réflexe interne. Nous présenterons alors le designer de cascades d'AAs qui sera notre niveau réflexe externe. Enfin, nous nous baserons sur des travaux réalisés dans le cadre du projet Continuum pour mettre en place les niveaux tactiques et stratégiques.

5.2.1 Une infrastructure logicielle à base de services pour dispositifs.

Nous avons vu que les architectures orientées services (SOA) sont largement utilisées en informatique mobile et ambiante pour représenter comme des services l'ensemble des fonctionnalités fournies par des dispositifs. Elles offrent des facilités pour l'interopérabilité entre les dispositifs, pour la découverte de services ou encore d'encapsulation (SECTION 2.2.2.1). Elles ont évoluées des SOA aux

1. http://continuum.unice.fr/

SOAD (architectures orientées services pour dispositifs) afin de s'adapter aux caractéristiques des dispositifs. Lorsque les technologies du Web sont utilisées pour implémenter les SOAD, l'interopérabilité entre tous les services est introduite. Seules deux implémentations des services web pour dispositifs existent aujourd'hui : UPnP [2] et DPWS [3]. Les SOAD ajoutent trois caractéristiques aux SOAD : (1) une découverte décentralisée réactive, (2) une gestion des apparitions et disparitions et (3) des communications événementielles.

5.2.1.1 Communications par événements

Les applications utilisant des dispositifs mobiles ont une nature réactive : les déconnexions fréquentes doivent être prises en compte rapidement. En effet, le fonctionnement sur batterie contraint les programmes à être efficaces et à utiliser des événements plutôt qu'un mécanisme de *polling*. Les capacités de traitement des dispositifs mobiles sont réduites en terme : de capacité de calcul, de quantité de mémoire et en durée d'utilisation à cause de l'alimentation par batterie. Les programmes doivent ainsi être les plus efficaces et pertinents possible. La mise en place de systèmes à file d'attente ou d'architectures complexes de publication/souscription [139], est rarement envisageable. C'est pourquoi des systèmes simples en relation $1 \rightarrow N$ sont utilisés.

Les communications par événements permettent aux applications basées sur des dispositifs d'utiliser de façon efficace les interruptions matérielles et offrent un découplage maximal. Si nous prenons l'exemple d'un interrupteur communiquant : lorsqu'il est actionné, une interruption matérielle est levée. Grâce aux événements, une notification sera immédiatement envoyée aux consommateurs. La dynamicité et l'efficacité sont maximales. Sans l'utilisation d'événements, l'interrupteur devrait garder une variable d'état à jour, en attendant que les consommateurs lui demandent sa valeur. Puisque ces invocations sont faites périodiquement, il existe un risque de manquer un changement d'état, s'il est plus court que la période d'invocation des clients.

Les communications par événements apportent la dynamicité nécessaire aux interactions entre les dispositifs prenant part aux applications d'informatique ambiante. Elles ajoutent de plus une dynamicité dans ces interactions, en permettant à tout moment aux entités concernées de les reconfigurer.

5.2.1.2 Apparition et disparition

Comme nous l'avons vu, en informatique ambiante, l'infrastructure d'une application est fortement variable en raison de la forte mobilité des entités qui la compose. La mobilité des utilisateurs, et donc des dispositifs de leurs réseaux personnels, provoque de fréquentes déconnexions et changements de réseaux. La topologie des réseaux ne peut pas non plus être connue à l'avance. Dans les réseaux mobiles, on ne peut pas connaître à l'avance l'adresse des annuaires, ni même supposer qu'il en existera un. Pour construire des applications reposant sur ces entités, un intergiciel doit connaître les entités se trouvant dans cette infrastructure. Les SOAD apportent une telle évolution aux SOA.

Les fournisseurs de services émettent des annonces par diffusion à tout le réseau lorsqu'ils y entrent (après adressage), ou lorsqu'ils sont en train de le quitter : cette apparition ou disparition par annonce est asynchrone, et fournit la propriété de dynamicité à cette infrastructure logicielle. Cependant, lorsqu'un consommateur arrive dans l'infrastructure, c'est à lui d'initier cette apparition

2. Universal Plug and Play Forum :`http://www.upnp.org/`
3. Device Profile for Web Services. `http://www.ws4d.org/`

ou disparition, sinon les fournisseurs ne pourront pas le découvrir rapidement. De plus, si les fournisseurs sont déconnectés brutalement, ou s'ils subissent une erreur de programmation, l'annonce de disparition ne sera pas envoyée. Pour palier à ces problématiques les fournisseurs utilisent un mécanisme de bail qui consiste à avertir régulièrement de leur présence.

A travers ces mécanismes d'annonces, les entités du réseau sont alors capables de connaître les entités se trouvant dans leur réseau dynamiquement et de les voir apparaître et disparaître. Ce mécanisme prend tout son sens lorsqu'il est couplé à une découverte dynamique décentralisée.

5.2.1.3 Découverte décentralisée réactive

Une autre évolution des SOA apportée par les SOAD est la modification de la découverte, pour prendre en compte des fournisseurs et consommateurs inconnus à l'avance. Les entités qui seront présentes pour créer une application ne sont donc pas connues, et doivent être découvertes de façon dynamique.

Des mécanismes de recherche décentralisée ont alors fait leur apparition, dans des standards industriels (SLP [140], Jini [133], Bluetooth SDP [4], Salutation [5], Bonjour [6]) et de nombreux projets de recherche ([160, 136, 157, 163, 88]). Ils permettent aux consommateurs de découvrir les fournisseurs sans passer par un registre centralisé. Le mécanisme de recherche est en fait ramené dans les fournisseurs et les consommateurs qui communiquent donc directement entre eux. La phase de découverte utilise alors les mécanismes d'apparition/disparition.

Ce mode est centré sur les consommateurs. Comme dans les architectures à services utilisant les registres, les consommateurs peuvent effectuer une requête de recherche, mais cette fois par diffusion. Dans la figure ci-dessous (FIG. 5.2), les traits en pointillés représentent des communications par diffusion, alors que ceux en trait plein sont point à point.

Toutefois, pour une meilleure efficacité ou sécurité, certains protocoles (SLP et Jini) permettent d'utiliser la diffusion pour découvrir un registre de services. Dans les environnements mobiles, les communications sans-fil sont souvent coûteuses en énergie. Le registre permet alors aux consommateurs d'avoir un seul interlocuteur et d'effectuer des requêtes en point à point. Un tel registre construit sa base de données à partir des messages d'annonce des fournisseurs.

La découverte est la première étape de la création d'applications à partir d'une infrastructure de services. En informatique ambiante, c'est d'autant plus le cas car les environnements dans lesquelles elles seront créées ne sont pas connus à priori. La découverte dynamique décentralisée couplée aux mécanismes d'apparition/disparition permet de découvrir les services sans les connaître *a priori* et sans se baser sur une entité statique.

4. Bluetooth Service Discovery Protocol, dans *Specification of the Bluetooth System. Core*, version 1.1. 2001.
5. Le *Salutation Consortium* n'existe plus.
6. Bonjour est utilisé dans Mac OS X pour découvrir des imprimantes et partager des données.

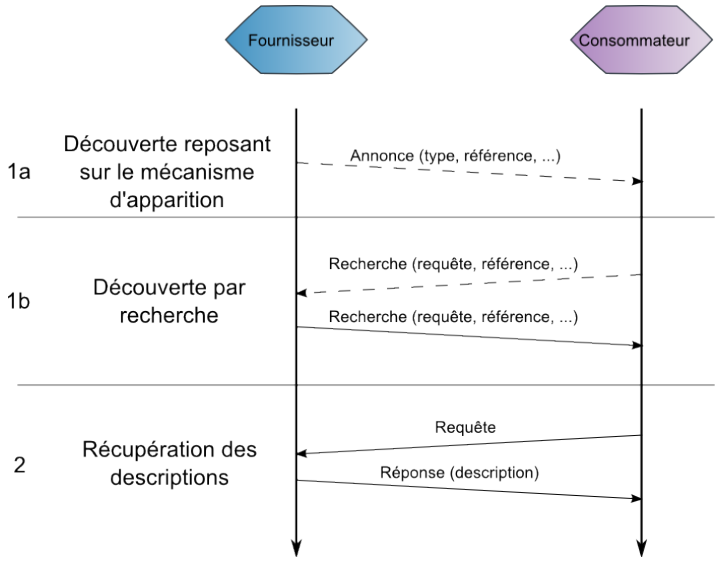


FIGURE 5.2 – Découverte dynamique décentralisée : protocoles de découverte et recherche.

5.2.1.4 Application au scénario

Dans le cadre de notre scénario, notre infrastructure reposera sur l'ensemble des dispositifs se trouvant dans le bâtiment intelligent (connus à l'avance) ainsi que de l'ensemble des dispositifs mobiles du personnel et des visiteurs (non prévisibles). Parmi ces dispositifs, nous trouvons : les fenêtres, l'ordinateur du gardien, des capteurs d'humidité, des capteurs de température, les PDA des techniciens ainsi qu'un capteur de niveau d'eau. Les capteurs d'humidité sont disséminés autour des fenêtres tandis que les capteurs de température sont dans le hall. Il permettront de gérer l'ouverture des fenêtres. Les capteurs pour le niveau d'eau se trouvent au sol et serviront à identifier si l'eau est au dessus d'un certain seuil dans le hall. Enfin, les dispositifs mobiles comme les téléphones portables ou les PDA n'étant pas connus à l'avance, les mécanismes d'apparition et de découverte dynamique sont pleinement utilisés afin que le système puisse les utiliser. Ce mécanisme nous permet de prendre en compte un premier challenge posé par ce scénario : la découverte dynamique et décentralisée nous permet de considérer la **variabilité** et l'**imprévisibilité** de l'infrastructure logicielle.

A partir de cette infrastructure de services pour dispositifs, nous souhaitons créer des applications en faisant interagir ces services vus comme des boites noires les uns avec les autres. Pour cela, dans le niveau réflexe interne de notre architecture, nous devons disposer d'une plateforme d'exécution permettant de faire ces orchestrations de services.

5.2.2 Niveau Réflexe interne : Plateforme d'exécution

Nous avons définis en SECTION 4.2.1 que la plateforme d'exécution associée à notre mécanisme d'exécution doit être une plateforme adaptative à base de composants. Plus particulièrement nous avons vu qu'il était intéressant d'utiliser une plateforme couplant les paradigmes à services et à composants de manière à pouvoir orchestrer des services par assemblage de composants dynamiquement. Dans cet optique, nous utilisons la plateforme WComp qui est une implémentation du modèle SLCA (*Service Lightweight Component Architecture*)[87]. Il s'agit d'un modèle d'architecture compositionnelle basée sur des événements, pour concevoir des assemblages de composants légers.

L'environnement est constitué d'utilisateurs mobiles interagissant avec le monde ou d'autres utilisateurs avec des dispositifs portés ou mobiles. Ils sont autant de services disponibles momentanément dans l'infrastructure. En générant et instanciant ou détruisant dynamiquement dans les assemblages des représentants vers ces services, et en utilisant des composants sur étagère qui permettent d'ajouter une logique applicative, les assemblages de composants offrent un moyen de créer des orchestrations de ces services.

5.2.2.1 Assemblages de composants légers

Ces composants sont dits « légers » pour plusieurs raisons. La première est qu'ils s'exécutent dans le même espace d'adressage et dans le même processus [65]. Leurs interactions peuvent être réduites au plus simple et au plus efficace : l'appel de fonction. La seconde raison, qui découle de la première, est qu'ils n'embarquent pas de code non-fonctionnel ou tout autre service technique inutile dans cet environnement local. Leur empreinte mémoire est donc réduite et ils sont instanciables et destructibles rapidement [113]. La granularité des composants légers est alors plus faible que celle des composants classiques. Un container n'est pas limité à un type de composants mais peut contenir tous les composants d'une application. Enfin, ils ne contiennent pas de références entre eux lors de la conception, et respectent les concepts de boîtes noires et de liaison tardive. Un pattern d'interaction asynchrone à découplage spatial est utilisé pour les liaisons entre les composants légers, favorisant le découplage et la réactivité.

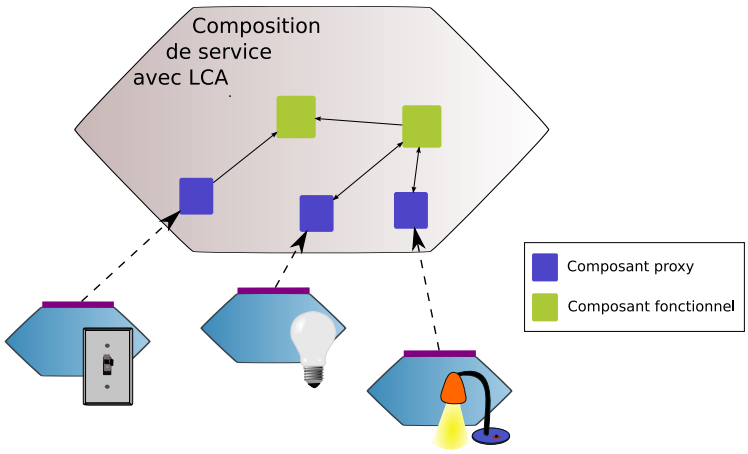


FIGURE 5.3 – Composition de services Web pour dispositifs avec des assemblages de composants légers LCA basés sur des événements.

La gestion des événements et des propriétés sont les seuls codes non-fonctionnels qui doivent être présents dans le code des composants. L'ajout de propriétés non-fonctionnelles, telles que la sécurité, la journalisation ou la persistance des messages peut se faire par ajout de composants dans l'assemblage, garantissant l'évolutivité du modèle.

Les composants SLCA possèdent une interface, définie par le type du composant. Cette interface est un ensemble de ports d'entrée : les méthodes, et de ports de sortie : les événements. Chaque port est typé par ses paramètres et possède un identifiant unique. Les interactions entre les composants sont les liaisons. Elles relient un port de sortie d'un composant à un ou plusieurs ports d'entrée.

Les ports étant explicites, aucun code n'a besoin d'être généré, ni étudié par introspection pour savoir à quel endroit modifier les composants pour changer la cible de sa liaison pendant l'exécution. Lorsque l'événement est envoyé par le composant qui est à la source de la liaison, le flot de contrôle passe alors aux composants destinataires, dans un ordre indéterminé, mais qui peut être fixé par des composants de séquence d'exécution. En se limitant à des liaisons uniques, et en utilisant des composants de séquence, la gestion des flots de contrôle de l'application est totalement déterministe. Le fait de ne pas avoir d'indirection due aux services techniques de la plate-forme donne un contrôle total aux composants sur les flots de contrôle, et facilite leur débogage.

La composition de services pour dispositifs par composants légers permet de créer de nouvelles applications dynamiques à partir des services présents dans l'infrastructure. Pour que les fonctionnalités créées localement soient réutilisables de l'extérieur il est alors possible d'exporter les fonctionnalités créées par un assemblage de composants vers l'infrastructure de services.

5.2.2.2 Distribuer la composition

SLCA permet de mettre en place des architectures à base de composants pour réaliser des orchestrations de services. Ces orchestrations peuvent elles-mêmes être encapsulées dans des services composites. Elles reposent sur un environnement d'exécution logiciel et matériel évoluant dynamiquement. Cet environnement est défini comme un ensemble de ressources, qui sont autant d'entités logicielles/physiques dont l'apparition et la disparition ne sont pas pilotées mais en général subies par l'application. SLCA est basé sur une infrastructure de services utilisant des événements, et découvrables dynamiquement de façon distribuée. Ils représentent les dispositifs utilisés dans les applications d'informatique ambiante, ainsi que les services composites créés par SLCA.

Les applications sont conçues par composition de services, en assemblant des composants. Un service composite est un container encapsulé dans un service pour dispositif contenant un assemblage de composants. Les composants *proxies* d'autres services Web sont donc instanciés dans le service composite et créent des applications à partir des services présents dans l'environnement. Un service composite peut créer une application communiquant avec un autre service composite. Il peut alors être vu comme une boîte grise, c'est à dire qu'il est possible d'en modifier l'assemblage et d'accéder à certaines fonctionnalités des composants de l'assemblage.

Un *service composite* fournit deux interfaces de services (FIGURE 5.4). L'interface fonctionnelle dynamique permet de publier et d'accéder aux nouvelles fonctionnalités fournies par le service composite. L'interface de contrôle, permet de modifier dynamiquement l'assemblage interne de composants SLCA qui fournit ces nouvelles fonctionnalités.

L'*interface fonctionnelle* permet d'exporter une application créée localement par l'assemblage de composants vers l'infrastructure de service. Les services composites participeront alors au graphe de coopération des services de l'infrastructure. L'interface est dynamique et exporte les événements et méthodes de l'assemblage interne en utilisant les *composants sondes*. L'ajout ou la suppression d'un composant sonde modifie dynamiquement l'interface fonctionnelle et le contrat du service composite correspondant. L'adaptation aux variations de l'environnement, en modifiant l'interface d'un service composite, est alors possible en cours d'exécution.

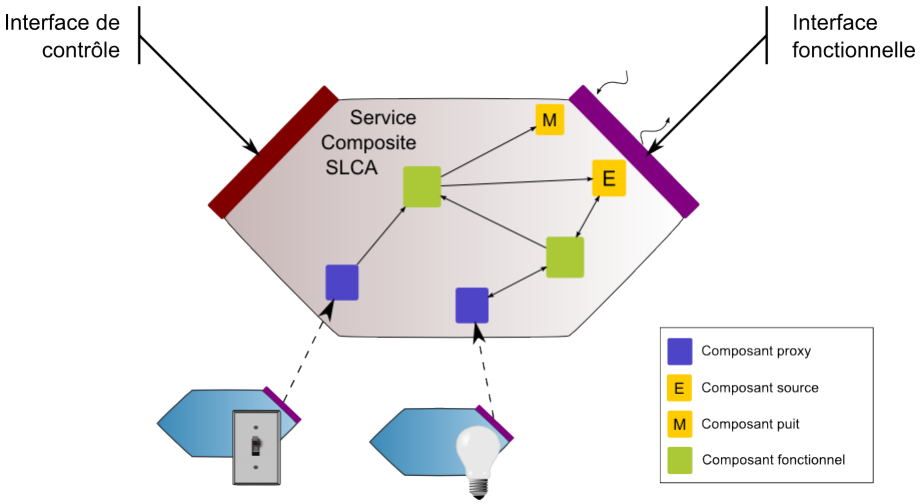


FIGURE 5.4 – Service Web composite basé sur les événements.

Deux types de composants sondes existent. Les *puits*, qui ajoutent une méthode à l'interface du service composite, et dans l'assemblage de composants interne, est un composant qui n'a qu'un port de sortie. L'invocation d'une méthode sur le service composite émet donc un événement dans l'assemblage de composants interne. Le deuxième type de composant sonde est la *source*, qui ajoute un événement à l'interface du service composite, et n'a qu'un port d'entrée. L'invocation d'une méthode dans l'assemblage de composants émettra un événement de service Web.

L'*interface de contrôle* permet de modifier dynamiquement l'assemblage interne du service. Elle fournit des méthodes pour ajouter ou supprimer des composants et des liaisons, ainsi que pour récupérer des informations sur les types de composants et l'assemblage. Ainsi, un autre client, qui peut être un service composite qui utilise un composant *proxy* pour le service, peut agir sur la structure d'un service composite. L'adaptation des services composites entre eux est alors possible. L'interface de service fournit aussi des événements, qui notifient les entités ayant souscrit aux changements structurels de l'assemblage interne du container. Ainsi, les adaptations peuvent être réactives. On peut alors imaginer toutes sortes de mécanismes d'adaptations, et outils d'aide à la création d'applications appelés *designers*.

5.2.2.3 Synchroniser l'application avec son infrastructure de services pour dispositifs

Les designers sont des consommateurs de service pour lesquels l'interface de contrôle des services composites est requise. Ils permettent de visualiser ou d'adapter les assemblages de composants des services composites en utilisant divers formalismes ou représentations. Parmi ces designers, nous pouvons citer l'*UPnPProxyDesigner*. Il s'agit d'un point de contrôle UPnP qui permet la découverte des services pour dispositifs se trouvant dans son infrastructure. Une fois ces dispositifs découverts, il permet, à travers l'interface de contrôle du service composite, la génération et l'instanciation de composants « proxy » dans le conteneur qui lui est associé. De la même manière, lors de la disparition d'un dispositif, il permet la suppression de son composant « proxy ». Ce designer fait le pont entre l'infrastructure et la plateforme d'exécution.

5.2.2.4 Application au scénario

La plateforme d'exécution nous permet de mettre en œuvre une orchestration à partir des services de l'infrastructure, aidé par des composants sur étagères. Dans le cadre du fonctionnement habituel du système, l'assemblage prend la forme présentée en FIGURE 5.5. Dans cet assemblage, l'ensemble

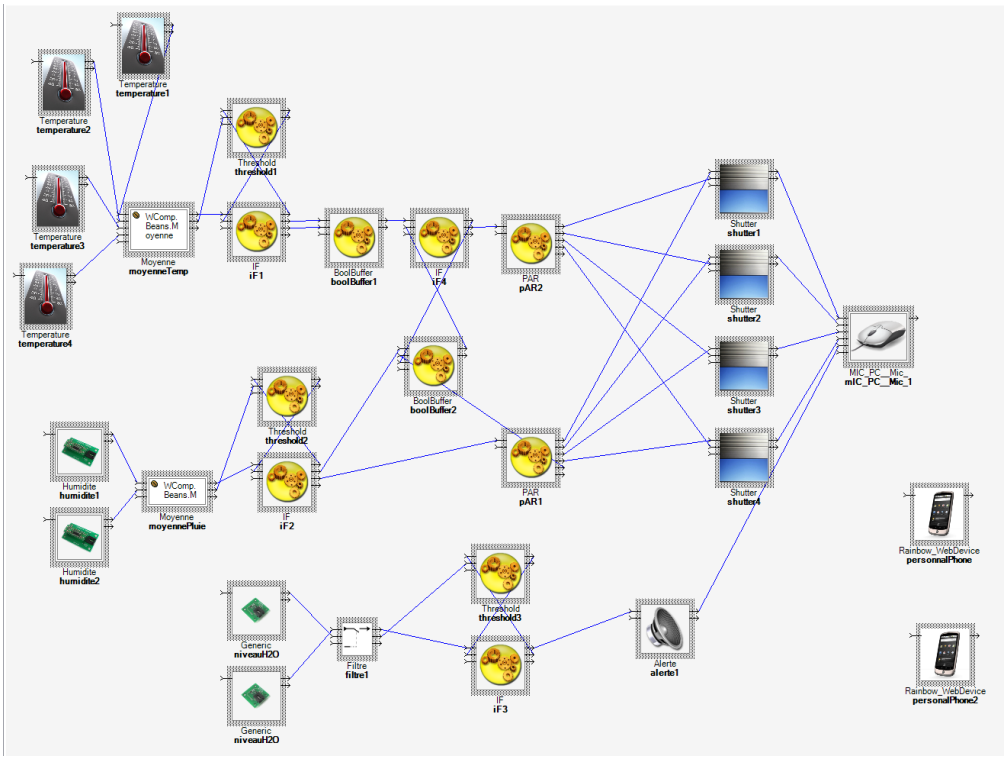


FIGURE 5.5 – Assemblage pour le fonctionnement habituel, sans alarme, du système.

des composants sans image ou dont l'image représente un engrenage, sont des composants sur étagères. Plus précisément, les composants avec les engrenages seront instanciés par les Cascades d'AAs, ils correspondent aux opérateurs du langage. Les composants sans image sont des composants génériques. Ils sont de deux types *Moyenne* et *BooleanBuffer*. Le premier type permet de faire une moyenne circulaire de plusieurs valeurs, tandis que le second possède un état (sous la forme d'un booléen) ; lorsque cet état change, une notification avec la valeur du booléen est émise. Pour le comportement de gestion des fenêtres via la température, ces composants seront utilisés pour faire (1) la moyenne des valeurs des capteurs de température et (2) filtrer les événements d'ouverture et de fermeture des fenêtres afin qu'ils interviennent uniquement sur changement d'état.

L'utilisation de cette plateforme nous permet d'apporter des solutions à une partie des challenges proposés par ce scénario : la plateforme nous permet de réaliser des **adaptations paramétrées** et fournit les mécanismes pour réaliser des **adaptations compositionnelles**. Il sera alors possible, à l'exécution, de faire **varier** l'assemblage de composants de manière **no-anticipée**. D'autre part, les**temps de réponse** offerts par la plateforme pour son adaptation sont faibles puisqu'ils sont de l'ordre de 0.048ms pour l'instanciation d'un composant.

La plateforme d'exécution et l'infrastructure logicielle fournissent le cadre technologique de base

pour la mise en œuvre de l'architecture. Le designer de Cascades d'AAs, qui est notre implémentation du niveau réflexe externe reposera sur ce cadre logiciel et sera lui-même implémenté à l'aide de ce cadre. Il prendra donc la forme d'un designer implémenté à l'aide de WComp.

5.2.3 Niveau Réflexe externe : Designer de Cascades d'AAs

Le designer de cascades d'AAs est lui-même implémenté à l'aide de WComp. Il utilise l'interface de contrôle d'un service composite pour ajouter ou retirer des composants ou liaisons dans l'assemblage applicatif.

5.2.3.1 Architecture du Designer de Cascades d'AAs

Ce designer se compose de deux services composites (FIGURE 5.7). Le premier joue le rôle d'interface tandis que le second est le tisseur à proprement parler. Le premier conteneur contient un pointeur vers le dépôt des cascades. Il permet à un utilisateur ou à un mécanisme externe, via l'interface fonctionnelle du service composite, de sélectionner les cascades à appliquer. Une fois les cascades sélectionnées, une chaîne de traitement allant du pointcut matching à la fabrique de greffon est déployée pour chaque AA qui la compose. Chacun de ces processus, que nous avons détaillés en SECTION 4.3.2, est implémenté sous la forme d'un composant. Au déploiement, lorsqu'une politique de combinaison est associée à un AA, le composant mettant en place cette dernière sera déployé. Par défaut, la politique de calcul de toutes les combinaisons est appliquée. Pour instancier ces composants, le service composite d'interface contient un proxy vers l'interface de contrôle du service composite du tisseur. Nous pouvons voir dans la FIGURE 5.7 que seule une cascade a été sélectionnée. Cette cascade contient trois AAs, par conséquent trois chaînes (visibles à gauche du conteneur 2) sont créées.

Le service composite qui implémente le tisseur est donc dynamiquement modifié en fonction des cascades sélectionnées. Il a pour objectif d'adapter le service composite applicatif. Pour ce faire, il dispose d'un composant proxy vers l'interface de contrôle du service composite. En plus des composants de la chaîne allant du pointcut matching à la fabrique de greffon, le composant *AdviceToUPnP* gère la synchronisation entre le modèle de l'application et l'application s'exécutant. Inversement, le composant *FormatingEntryBean* et le composant proxy vers le service applicatif gèrent la synchronisation entre l'application et son modèle. Ce composant est notifié via l'interface de contrôle du service applicatif de toute modification compositionnelle intervenant dans le conteneur applicatif. De plus, il gère le modèle de l'application intermédiaire c'est-à-dire représentant le résultat de l'adaptation entre les différents cycles de tissages. Le composant *dispatcher* a pour objectif de gérer le bon ordonnancement des cycles de tissage et de transmettre le modèle de l'application sur lequel le cycle doit être réalisé (cf. SECTION 4.4.3.2). Enfin les composant *superposition* et *conflict manager* réalisent les processus de superposition et de fusion.

5.2.3.2 Application au scénario

Nous exprimons les comportements déployés par le niveau tactique avec la décomposition classique des boucles d'auto-adaptation : perception, décision et action. Cette décomposition nous permet de gérer finement la portée des différents aspects composant les cascades et offre la capacité à un aspect de réutiliser des éléments instanciés par d'autres aspects. Nous utiliserons ce points pour

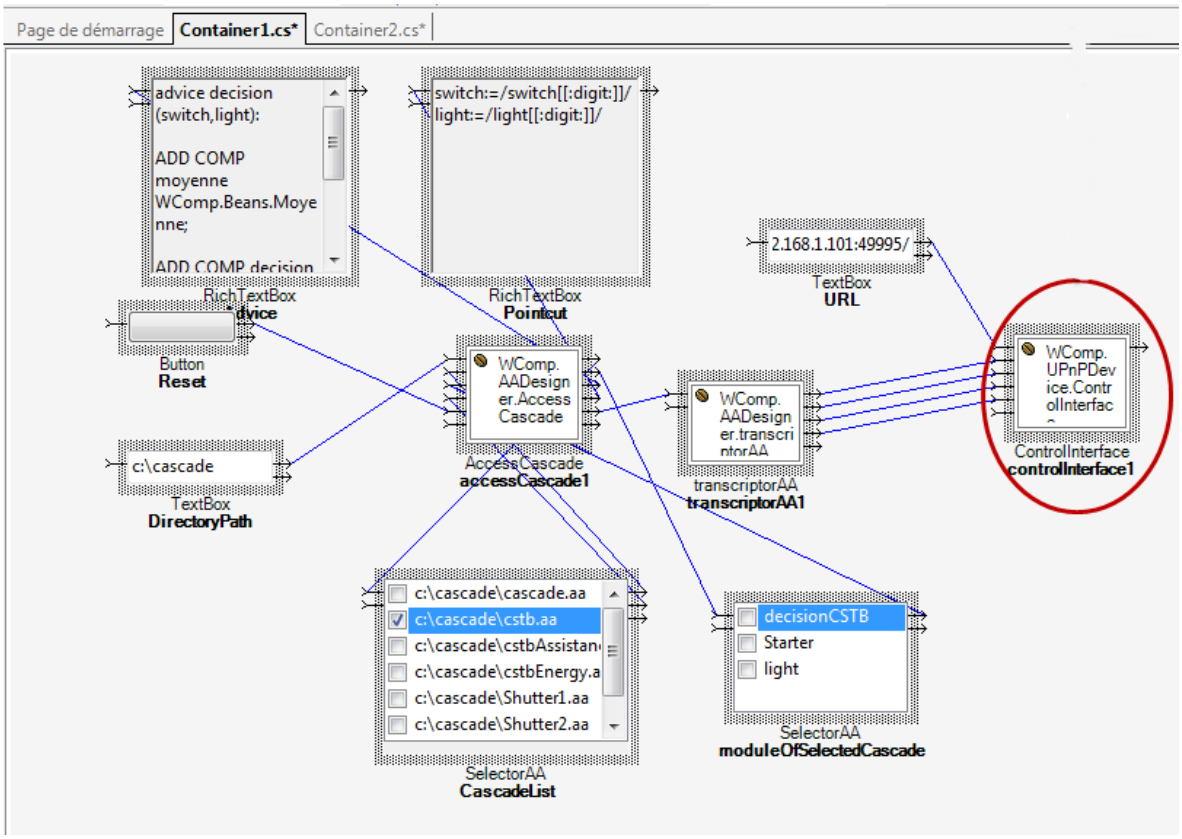


FIGURE 5.6 – Le conteneur constituant l'interface du designer de cascades d'AAs.

les parties de décision et de perception, par exemple afin de pouvoir agréger et analyser en un point les données provenant de plusieurs capteurs (cf. SECTION 4.2.3.4). D'autre part, cette décomposition nous permet de réduire le nombre de règles d'adaptation à écrire grâce à une forte réutilisation des aspects dans les cascades.

Nous allons étudier plus précisément ici, les cascades permettant de décrire les comportements pour la gestion des fenêtres qui sont les plus complexes. Les autres comportements sont présentés en ANNEXE **??**. Ces deux cascades sont indépendantes et prennent la forme suivante : *CascadeFenetreTemp={{moyenneTemp},{calcul_moyenneTemp},{gestion_fenetreTemp}}* et *CascadeFenetrePluie={{moyennePluie},{calcul_moyennePluie},{gestion_fenetrePluie,gestion_fenetrePluiePrioritaire}}*. Les aspects sont donc répartis dans les différents cycles comme présenté en FIGURE 5.9.

Dans ces cascades, les deux aspects pour le premier cycle de tissage (FIGURE 5.10 et FIGURE 5.11) sont simples et instancient des composants de type *Moyenne* et *BooleanBuffer*.

Les deux aspects de ces cascades, destinés au second cycle de tissage, consistent à relier les capteurs aux composants de moyenne qui leurs sont associés. La politique de combinaison qui leur est associée est de type *n :n*.

Les AA destinés au troisième cycle de tissage sont les plus complexes et mettent en jeu le

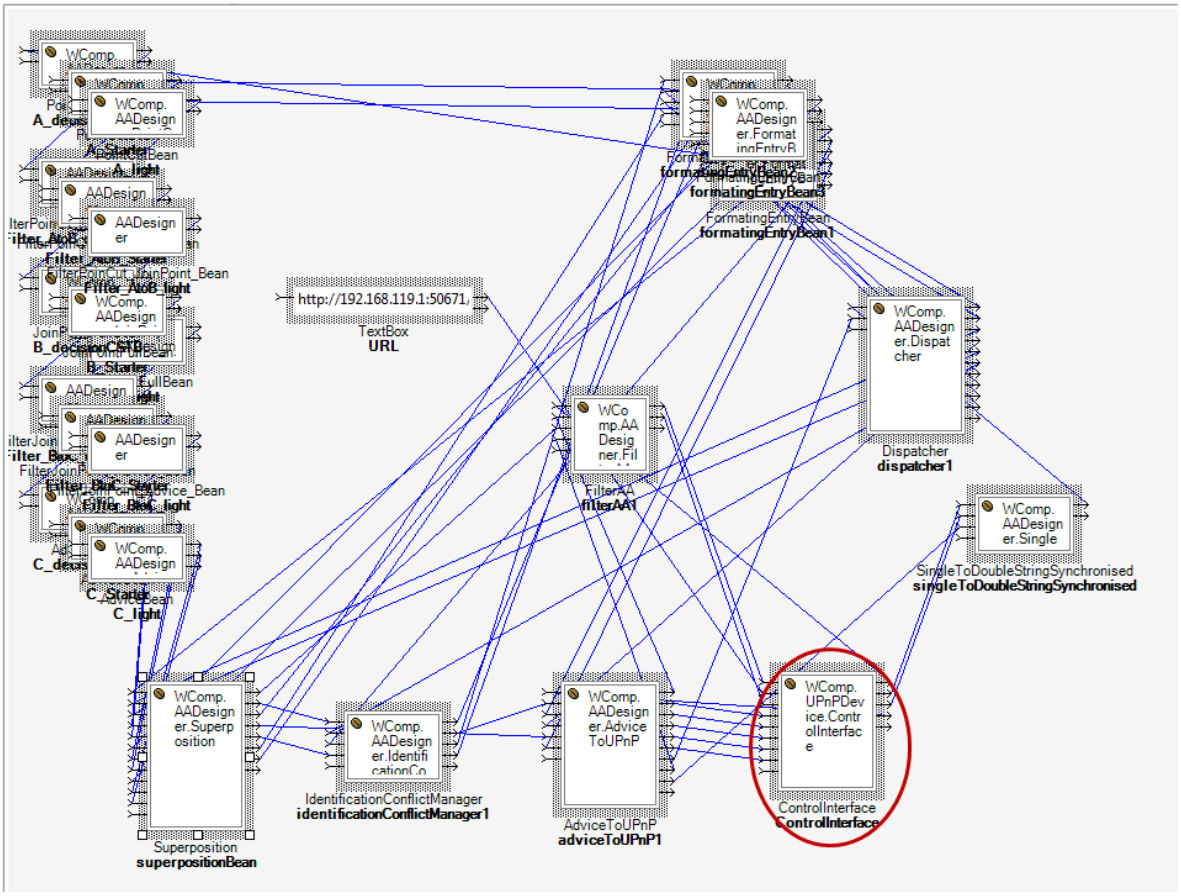


FIGURE 5.7 – Le conteneur constituant le tisseur du designer de cascades d'AAs.

mécanisme de fusion et des opérateurs du langage. L'AA présenté en FIGURE 5.14 met en place le système d'ouverture des fenêtres en fonction de la moyenne des températures observées. Un composant seuil (*threshold*) est instancié afin de déterminer si la température est suffisante pour ouvrir la fenêtre. Le comportement décrit à la ligne 10 est alors le suivant : lors d'une variation de la température (événement provenant de moyenne), si le seuil est dépassé alors la demande d'ouverture est transmise au *BooleanBuffer*, sinon il s'agit d'une demande de fermeture. Le *BooleanBuffer*, lorsqu'il change d'état, déclenche alors sur les fenêtres l'ordre d'ouverture ou de fermeture en fonction de son état.

Les AAs destinés au troisième cycle de tissage, pour le comportement de gestion des fenêtres en fonction de la pluie, dans une approche semblable aux précédents, ils peuvent être écrits comme en FIGURE 5.15 et 5.16. Le premier (FIGURE 5.15) met en place le mécanisme suivant : si la moyenne des capteurs d'humidité est supérieure à un certain seuil, alors dans un premier temps les fenêtres sont fermées (*fenetre.Close*) et un *BooleanBuffer*, qui servira aux autres mécanismes d'ouverture à identifier s'ils sont autorisés à ouvrir les fenêtre est placé à faux. Sinon il est placé à vrai.

Le second AA (FIGURE 5.16) utilisera ce *BooleanBuffer* afin d'ajouter, à tous les mécanismes permettant de gérer l'état des fenêtres à l'aide de l'opérateur *call*, un contrat dont le comportement est le suivant : la gestion des fenêtres est autorisée si et seulement si il ne pleut pas.

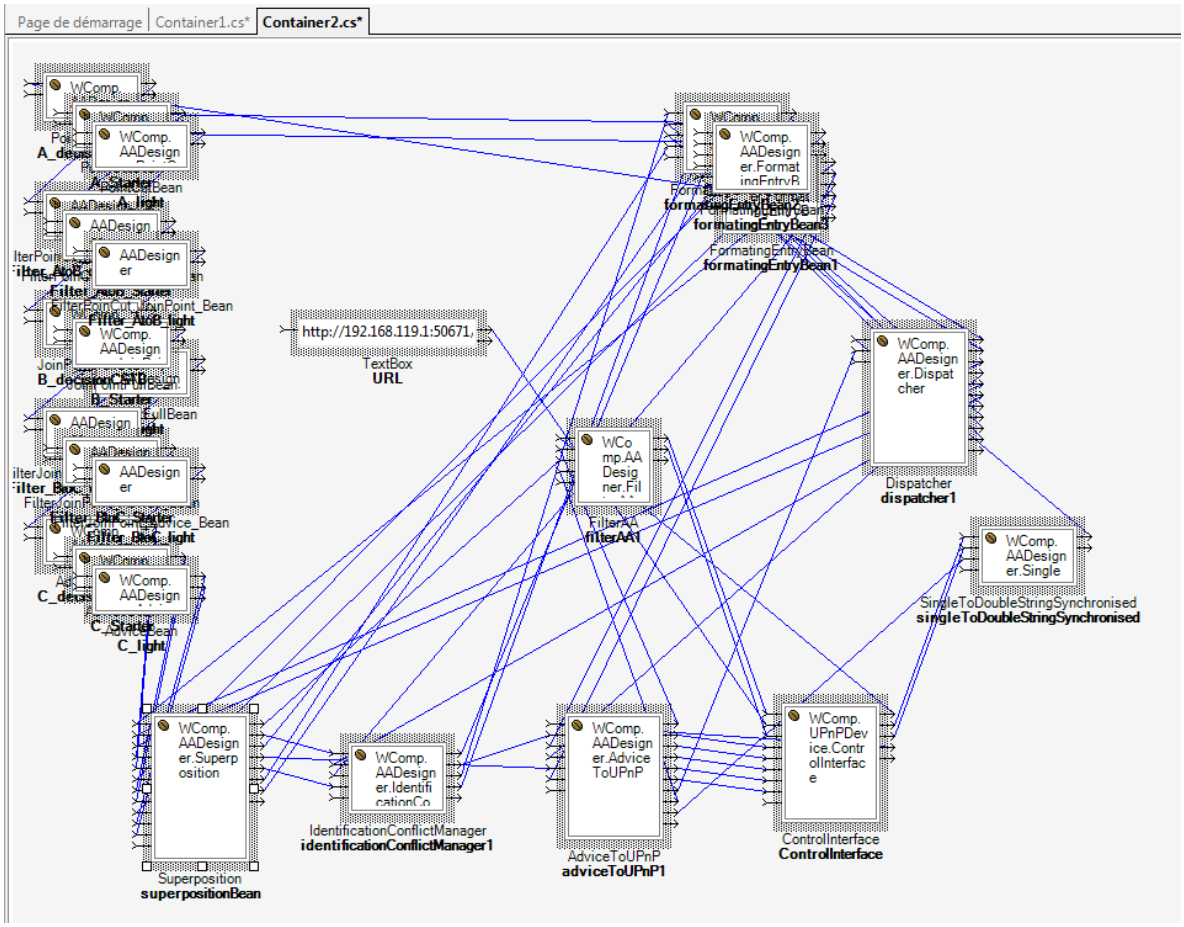


FIGURE 5.8 – Le tisseur de cascades d'AAs.

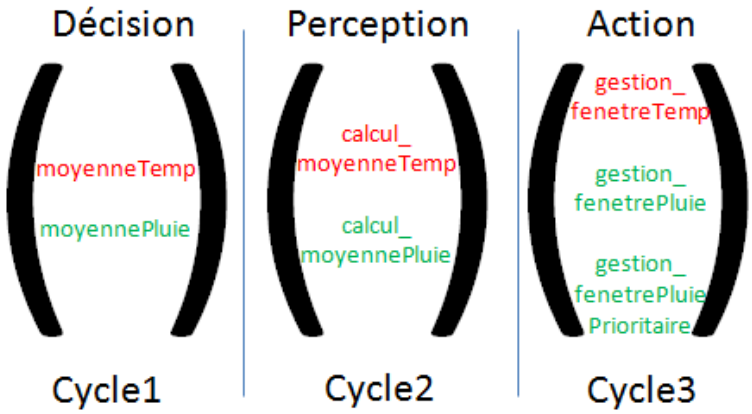


FIGURE 5.9 – Répartitions des AAs dans les cycles de tissages

Lorsque les techniciens arriveront sur les lieux, ces deux comportements seront retirés. Dans le cas de fonctionnement habituel du système l'assemblage de composants déployé au niveau réflexe

```
Advice:
  schema moyenneTemp () :
        MoyenneTemp:BasicBean.Moyenne
        BBufferTemp:BasicBean.BooleanBuffer
```

FIGURE 5.10 – Partie décision du comportement de gestion des fenêtres en fonction de la température.

```
Advice:
  schema moyennePluie () :
        MoyennePluie:BasicBean.Moyenne
        BBufferPluie:BasicBean.BooleanBuffer
```

FIGURE 5.11 – Partie décision du comportement de gestion des fenêtres en fonction de la pluie.

```
Pointcut:
  temp:=/temperature*.^Evented_NewValue/
  moyenne:=/MoyenneTemp.*/

Advice:
  schema calcul_moyenneTemp (temp,moyenne) :
        temperature -> moyenne.SetValue
```

FIGURE 5.12 – Partie perception du comportement de gestion des fenêtres en fonction de la température.

```
Pointcut:
  pluie:=/humidite*.^Evented_NewValue/
  moyenne:=/MoyennePluie.*/

Advice:
  schema calcul_moyennePluie (pluie,moyenne) :
        pluie -> moyenne.SetValue
```

FIGURE 5.13 – Partie perception du comportement de gestion des fenêtres en fonction de la pluie.

```
Pointcut:
        fenetre:=/fenetre[[:digit:]].*/
        moyenne:=/MoyenneTemperature.^AverageEvent/
        BBufferTemp:=/BBufferTemp.*/
Advice:
        schema gestion_fenetreTemp (fenetre,moyenne,BBufferTemp) :
                threshold: BasicBeans.Threshold(threshold=10)

                moyenne -> if(threshold.IsReached){BBufferTemp.SetTrue}
                        else{BBufferTemp.setFalse}
                moyenne -> threshold.SetValue
                BBufferTemp.^StateChange -> fenetre.SetState
```

FIGURE 5.14 – Partie action du comportement de gestion des fenêtres en fonction de la température.

interne prendra la forme présentée en FIGURE 5.5.

Le designer de Cascade nous permet de répondre à une partie des challenges proposés par notre scénario : **extensibilité** de l'ensemble des adaptations à l'exécution ; les **dynamiques sont séparées** entre l'application et le mécanisme d'adaptation ; l'ensemble des adaptations variables et la prise en compte de la **variabilité** de l'infrastructure ; l'ensemble des adaptations et la manière dont elles sont composées est **non-anticipée**. D'autre part, les **temps de réponse** offerts par ce mécanisme d'adaptation sont adaptés, dans le cadre de notre scénario, le temps de calcul des adaptations est de l'ordre de 200ms. La gestion des comportements et donc des cascades d'AAs à déployer revient au niveau tactique. Ce dernier, à travers l'interface fonctionnelle du service composite d'interface du designer de peut sélectionner et déployer de nouveaux comportements.

```
Pointcut:
        fenetre:=/fenetre[[:digit:]].*/
        moyenne:=/MoyennePluie.^AverageEvent/
        BBufferPluie:=/BBufferPluie.*/
Advice:
        schema gestion_fenetrePluie (fenetre,moyenne,BBufferPluie) :
                threshold: BasicBeans.Threshold(threshold=10)

                moyenne −> if(threshold.IsReached)
                        {fenetre.Close ; BBufferPluie.SetTrue} else {BBufferPluie.setFalse}
                moyenne −> threshold.SetValue
```

FIGURE 5.15 – Partie action du comportement de gestion des fenêtres en fonction de la pluie.

```
Pointcut:
        fenetre:=/fenetre[[:digit:]].*/
        BBufferPluie:=/BBufferPluie.*/
Advice:
        schema gestion_fenetrePluiePrioritaire (BBufferPluie,fenetre) :
                fenetre.SetState −> (if (BBufferPluie.GetState) call)
```

FIGURE 5.16 – Partie action prioritaire du comportement de gestion des fenêtres en fonction de la pluie.

5.2.4 Niveau Tactique : Gestionnaire de contextes

Dans le niveau tactique, dans l'optique de déployer les cascades d'AAs, nous utilisons les travaux que nous avons réalisés sur un gestionnaire de contextes dans le cadre du projet Continuum [105]. L'objectif de ce gestionnaire de contexte est d'identifier une situation courante et de déployer en fonction de cette dernière les cascades d'AAs adéquates. Les communications entre le designer de cascade et ce gestionnaire se font également via UPnP. Ce gestionnaire de contexte repose sur le processus de modélisation du contexte à la conception définie dans [105]. Dans ce processus, un concepteur défini un ensemble d'entités considérées comme intéressantes et un ensemble de prédicats portant sur une partie de ces entités. De cette manière, chaque contexte, en fonction des prédicats validés, peut prendre la forme d'un automate dans lequel chaque état correspond à une situation. On se trouve dans une situation lorsque l'ensemble des prédicats qui lui sont associés sont vérifiés. On change de situation lorsqu'il y a un changement dans l'ensemble des prédicats vérifiés.

Dans le gestionnaire de contexte, chaque contexte prend la forme d'un fichier XML appelé *contextSet*. Il peut y en avoir N. Un *contextSet* contient un ensemble de cascade d'AAs qui devra être déployé dans le cas ou le contexte est validé. Des contraintes peuvent être exprimées entre ces cascades comme, par exemple, une contrainte d'exclusion entre plusieurs cascades. Enfin, un contexte contient la définition de prédicats ainsi qu'une liste de situations auxquelles sont associées les prédicats définis et les valeurs attendues de ces prédicats pour valider la situation. Par exemple, la *situation A* est identifiée comme courante quand le *prédicat 1* est à faux et le *prédicat 2* à vrai. Dans le cadre de continuum, les prédicats sont des requêtes SPARSQL qui sont adressées à une base de données.

Le niveau stratégique a pour objectif d'ajouter dans le gestionnaire de contexte de nouveaux *contextSet*. Lorsqu'un nouveau contextSet est déployé, dans un premier temps, sa cohérence avec le schéma XSD est vérifiée ainsi que les prédicats, situations et AAs. Une fois les *contextSet* validés et donc déployés, les prédicats sont évalués. Une fois ce processus réalisé, les situations courantes sont identifiées et les listes de cascades sont fusionnées en vue d'être déployées sur le designer de cascades.

5.2.4.1 Application au scénario

Dans cette architecture, pour la mise en place de notre scénario, nous identifions deux situations avant l'intervention des techniciens : (1) fonctionnement normal et (2) inondation. Le fonctionnement normal met en jeu trois comportements :

1. **Comportement de gestion de l'ouverture des fenêtres en fonction de la température :** ce comportement gère l'ouverture et la fermeture des fenêtres en fonction de la température.
2. **Comportement de monitoring de l'état des fenêtres :** ce comportement est destiné au gardien et lui permet de suivre l'état des fenêtres.
3. **Comportement d'alerte du gardien :** ce comportement est toujours déployé car le dispositif d'affichage est connu à l'avance. D'autre part, il est important d'envoyer ce message d'alerte au plus vite, ce que permet l'adaptation paramétrée.
4. **Comportement de gestion de l'ouverture des fenêtres en fonction de la pluie :** ce comportement gère la fermeture des fenêtres en fonction de la pluie. Il est prioritaire sur celui de température.

La seconde situation qui intervient en cas d'inondation met en jeu cinq comportements. Aux comportements précédents, est ajouté celui d'**alerte du personnel**. Une fois les techniciens sur les lieux de l'intervention, ils créent une nouvelle situation qui consiste à retirer les comportements de gestion des fenêtres et à ajouter un **comportement de contrôle des fenêtres**. Le comportement qu'ils déploient, qui n'avait pas été anticipé auparavant, s'intègre alors avec ceux existants. Le gestionnaire de contexte contient alors les situations présentées en FIGURE 5.17.

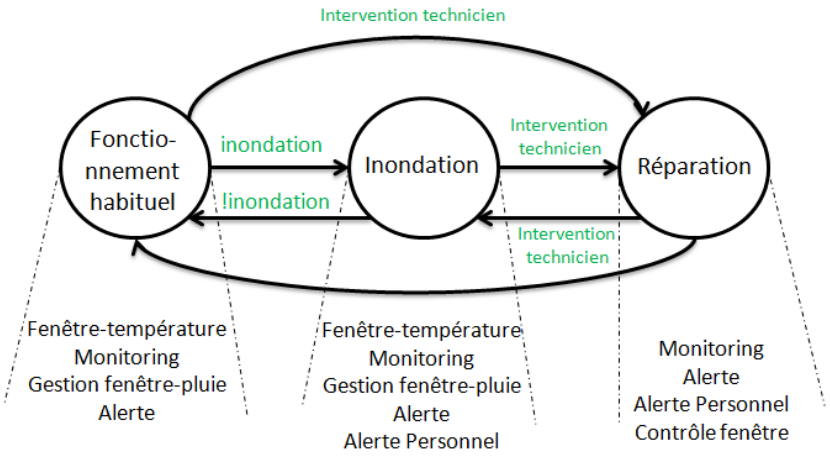


FIGURE 5.17 – États dans le gestionnaire de contexte

Lorsqu'une situation d'inondation se présente, le comportement d'alerte au personnel est déployé, l'assemblage évolue alors comme nous pouvons le voir dans la FIGURE 5.18.

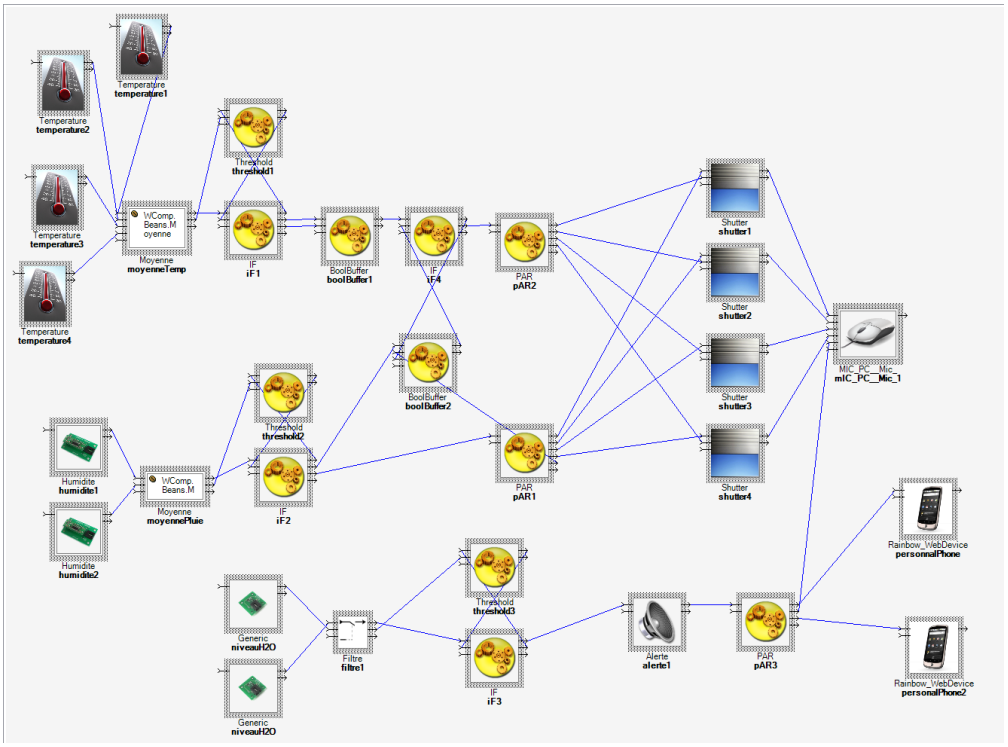


FIGURE 5.18 – Assemblage de composants dans la situation d'inondation.

Enfin, après avoir été déployée par les techniciens, lorsque la troisième situation survient, les comportements de gestion des fenêtres sont retirés tandis qu'un comportement pour les contrôler via leur téléphone est déployé. Cette situation mène à l'assemblage décrit en FIGURE 5.19.

Le gestionnaire de contexte nous permet de gérer les différentes situations se trouvant dans notre scénario avec une dynamique séparée de celle du designer de Cascades d'AAs. Les modifications de ce niveau sont réalisées via le niveau stratégique. Nous n'entrerons pas dans les détails de ce niveau. Cependant, puisque tourné vers l'utilisateur, ce niveau doit disposer de mécanismes qui lui permettant d'intervenir sur le comportement du système.

5.2.5 Niveau Stratégique

Nous n'entrerons pas dans les détails de la description du niveau stratégique. Dans le cadre du projet Continuum, des travaux ont portés sur la définition d'une grammaire d'un langage pseudo-naturel [79] défini par l'équipe IIHM du LIG. Ce langage, s'adresse aux utilisateurs finaux. Il permet à la fois la génération de contexte et de cascades d'AAs. D'autres approches pourraient être utilisées, par exemple, Relax [162] propose, aux concepteurs, un langage facilitant la conception de « requirements » pour les systèmes auto-adaptatifs. Ce langage adresse explicitement la problématique de l'imprévisibilité dans la spécification des requirements avec un formalisme exprimé en termes de logique floue. Dans le cadre de Relax et appliqué au scénario, un requis simple pourrait être défini

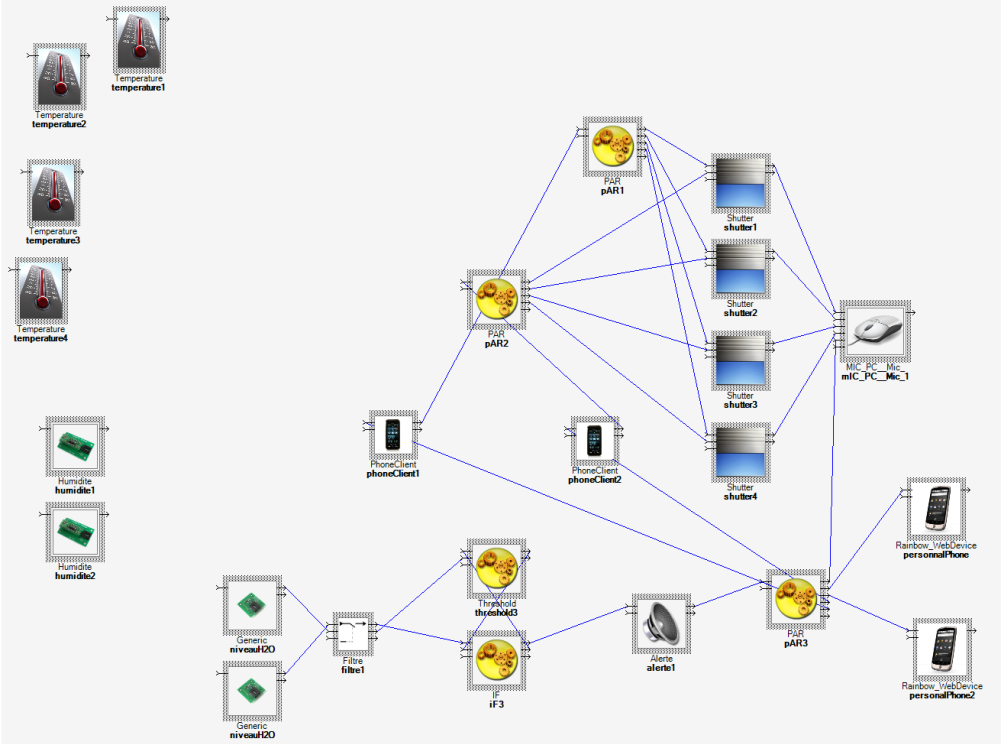


FIGURE 5.19 – Assemblage de composants dans la situation de réparation.

comme en FIGURE 5.20.

R1: Humidity sensor SHALL detect rain **and close** AS MANY windows AS POSSIBLE
MON: humidity sensor
ENV: humidité
REL: le capteur d'humidité informe sur l'humidité extérieur (pluie ou non)

FIGURE 5.20 – Définition d'un requis à l'aide de Relax.

5.3 Conclusion

Dans ce chapitre, nous avons implémenté notre modèle architectural à partir d'un cadre de travail basé sur la plateforme d'exécution WComp et une infrastructure logicielle de services pour dispositifs. A partir de ce cadre, nous avons pu implémenter notre niveau réflexe externe comme un designer de Cascades d'AAs (lui-même réalisé avec WComp) puis notre niveau tactique à l'aide du gestionnaire de contexte. Nous avons ainsi spécifier les différentes situations de notre scénario et les comportements qui leur sont associés. Ces comportement sont définis comme des cascades d'AAs. Enfin, nous avons présenté plusieurs pistes qui permettraient la réalisation du niveau stratégique.

Quatrième partie

Conclusions et perspectives

CHAPITRE 6

Conclusions et perspectives

« Ce n'est pas la fin. Ce n'est même pas le commencement de la fin.
Mais, c'est peut-être la fin du commencement. »

Winston Churchill.

Sommaire

DANS ce chapitre nous ferons une synthèse des travaux présentés dans cette thèse puis nous identifierons quelques perspectives de poursuite de ces travaux.

6.1 Synthèse

Nous avons observé, à travers l'état de l'art, que les mécanismes de décision, dont l'objectif est de sélectionner les adaptations qui doivent être réalisées dans une situation donnée, spécifient également « comment » les composer dans des approches plus ou moins rigides. Ces approches sont plus ou moins apte à prendre en compte l'imprévisibilité telle que nous l'avons définie (SECTION 1.2.2). Ces mécanismes de décision se trouvent classiquement dans des architectures centralisées, découpés comme une séquence de processus utilisant une représentation interne de leur environnement. Il leur est difficile de suivre plusieurs dynamiques. Les traitements, ainsi que la maintenance de cette représentation, peuvent mener à une désynchronisation entre l'environnement et sa représentation.

Dans cette thèse, nous avons étudié une architecture et un mécanisme d'adaptation pour l'adaptation des applications en informatique ambiante avec des propriétés logiques et temporelles. Nous avons proposé un modèle architectural qui offre un cadre de travail pour la conception de système ambiant. Ce modèle architectural permet de concevoir des systèmes ambiants en mesure de prendre en compte, avec diverses dynamiques, l'évolution de leur environnement, grâce à une décomposition en

quatre niveaux. Ce modèle propose d'organiser, dans ses niveaux, différents mécanismes d'évaluation de cet environnement. Les traitements réalisés dans chaque niveau étant indépendants les uns des autres, lorsqu'un processus de prise en compte du contexte complexe est lancé, les processus plus simples restent toujours en fonctionnement tout comme l'application. La continuité de service est maximale.

Nous avons associé au modèle architectural un mécanisme d'adaptation qui peut être utilisé à la fois par les mécanismes de décision les plus rigides comme les plus flexibles. Nous avons vu que la mise en œuvre de ce mécanisme d'adaptation à l'aide de cascades d'aspects respecte un ensemble de propriétés logiques et temporelles. Ces propriétés garantissent : l'indépendance des entités d'adaptations ; l'extensibilité à l'exécution de l'ensemble des règles d'adaptations sans se préoccuper de celles existantes ; une capacité à composer des entités d'adaptations sans que cela soit anticipé à la conception ; une application et composition opportuniste des adaptations. La mise en œuvre de ce mécanisme à l'aide des Aspects d'Assemblages nous a permis de vérifier que les temps de réponse proposés par l'approche sont adaptés et maîtrisés.

Enfin, nous conclurons cette synthèse en positionnant de quelle manière les travaux réalisés dans cette thèse permettent de répondre aux questions que nous avons soulevées en introduction.

- *Comment définir un mécanisme d'adaptation permettant d'exprimer la grande variabilité du système tout en minimisant le nombre d'entité d'adaptations à écrire ?*

⇒ Afin de permettre l'expression d'une grande variabilité du système avec un maximum de réutilisation, le mécanisme d'adaptation proposé repose sur une décomposition comportementale hybride. Il s'agit d'une décomposition fonctionnelle dans une décomposition comportementale. La mise en place de ces comportements hybrides repose sur une approche inspirée de celles orientées aspects et features appelée cascade d'aspects. Il est alors possible d'exprimer l'adaptation à la fois comme une préoccupation transverse homogène et hétégogène.

- *Comment associer aux différents mécanismes de décision un mécanisme d'adaptation que tous peuvent utiliser, c'est-à-dire permettant de déployer des adaptations sans se préoccuper de celles existantes ou de les combiner explicitement ?*

⇒ Le mécanisme d'adaptation proposé repose sur les cascades d'aspects. Grâce à la symétrie de l'opération de tissage des cascades, les adaptations sont indépendantes les unes des autres mais aussi auto-suffisantes. Un système, peut donc déployer des cascades sans se préoccuper de celles qui le sont déjà. De cette manière, divers acteurs ont la possibilité de déployer au runtime leurs cascades dans un même tisseur distribuant et décentralisant ainsi les règles d'adaptation parmi les acteurs. D'autre part, les cascades permettent d'exprimer un ordre ou une absence d'ordre entre des adaptations. Plus encore, les cascades permettent d'exprimer ces deux notions dans un même entité, comportement, d'adaptation. Elles peuvent ainsi être utilisées à la fois par des mécanismes de décision fortement contraint qui requièrent un ordre entre les adaptations et par des mécanismes plus fléxibles capables de prendre en compte une certaine imprévisibilité.

- *Comment permettre le déclenchement des adaptations indépendamment les unes des autres ?*

⇒ Un designer ne peut connaître à priori l'ordre dans lequel ces adaptations vont être tissées. En effet, ce dernier ne peut connaître l'ordre dans lequel vont apparaître/disparaître des composants dans l'assemblage. D'autre part, un acteur ne peut connaître a priori les autres adaptations déployées. La manière dont les entités d'adaptations doivent être composées ne peut donc être anticipée. Les cascades et aspects permettent, grâce au respect de la propriété de symétrie de

l'opération de tissage, d'avoir une composition non-anticipée. Elles peuvent ainsi être déclenchées indépendamment les unes des autres.

- *Comment prendre en compte les multiples dynamiques d'évolution de l'environnement ?*

⇒ Nous avons présenté une architecture sur quatre niveaux qui propose un cadre de travail pour la conception de systèmes ambiants et donc sensibles au contexte. Grâce à l'indépendance des niveaux, leur possible décomposition comportementale et à la hiérarchie établie entre eux (basée sur leurs temps de réponse ainsi que leur gestion de l'état de l'environnement), cette architecture permet de respecter les diverses dynamiques d'évolution de l'environnement. Ce qui permet de garantir la pertinence temporelle de l'adaptation tout en autorisant le maintient de la pertinence logique du système.

- *Comment organiser les mécanismes de décision ?*

⇒ L'architecture sur quatre niveaux est suffisamment générique pour permettre l'utilisation en son sein des différentes approches de décision et d'adaptation que nous avons vu dans l'état de l'art. Nous avons regroupés les différentes combinaisons d'approches de décision et d'adaptation en quatre catégories déployables dans les quatre niveaux. Les différentes combinaisons sont répartis dans les différents niveaux en fonction de plusieurs critères : leurs temps de réponse, leurs gestion de l'état de l'environnement, et l'extensibilité de l'adaptation. En particulier, plus les temps de réponse sont faibles, plus les mécanismes sont proches de l'application.

- *Comment prendre en compte ces multiples évolutions en interrompant le moins possible le système ambiant ?*

⇒ L'architecture sur quatre niveaux apporte une première solution à cette problématique. Dans cette dernière, les traitements étant indépendants les uns des autres, lorsqu'un processus de prise en compte du contexte complexe (par exemple dans les niveaux tactiques ou stratégiques) est lancé, les processus plus simples (réflexes) restent toujours en fonctionnement et l'application également. La continuité de service est maximale. Le mécanisme d'adaptation offre une seconde solution à cette question. Grâce à l'utilisation d'une approche de type *models@runtime*, l'application n'est adaptée que lorsque l'ensemble des modifications qui doivent lui être portées ont étés calculées et vérifiées.

L'ensemble de ces travaux de thèse ouvre de nombreuses perspectives pour des travaux futurs que ce soit dans le domaine de l'informatique ambiante ou dans d'autres domaines comme le Web sémantique ou encore l'AOP. Nous terminerons ce chapitre par une présentation succinte de quelques unes de ces perspectives.

6.2 Perspectives de recherche

Nous allons présenter dans cette section un ensemble de perspectives aux travaux présentés dans cette thèse qui nous semblent intéressantes à explorer.

6.2.1 Faire évoluer les points de coupes des AAs

Afin de gérer plus finement le tissage des Aspects d'Assemblage, leurs points de coupe peuvent être étendus dans l'optique de ne plus considérer uniquement les méta-données associées aux composants. Par exemple, des points de coupes sémantiques permettraient de prendre en compte dans

le processus d'adaptation l'hétérogénéité sémantique des dispositifs qui composent l'infrastructure logicielle d'une application. Une autre possibilité serait de leur permettre d'identifier des équivalences sémantiques ou de QoS entre les services de l'infrastructure afin d'autoriser la substitution de services par d'autres équivalents. Des travaux comme ceux présentés dans [89] pourraient alors s'appliquer.

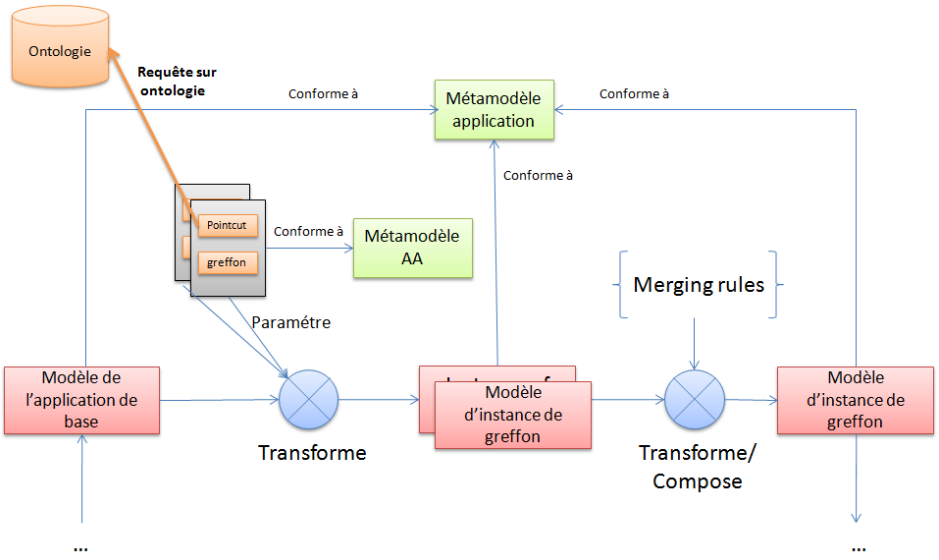


FIGURE 6.1 – Vers des points de coupe sémantiques

Une autre particularité des points de coupes des AAs est leur application opportuniste ; il pourrait être intéressant d'intégrer d'autres politiques. Par exemple, une possibilité serait de retirer, de manière opportune, les fonctionnalités reposant sur des dispositifs qui viennent de quitter l'infrastructure logicielle d'une application, et, par contre, de ne pas tisser automatiquement les aspects qui peuvent l'être. Une telle politique permettrait d'augmenter la stabilité du système, les modifications pour amélioration du système pourraient par exemple être décidées par l'utilisateur une fois que celui-ci les juges opportunes. Il faudrait alors distinguer parmi les perturbations, les disparitions des apparitions de dispositifs.

6.2.2 « Model checking » sur les applications en entrée et sortie du tisseur

Il pourrait être intéressant de vérifier à l'aide de model checking si un modèle d'application passé en entrée du tisseur de cascades vérifie un ensemble de spécifications. Ces vérifications pourraient permettre de personnaliser le tisseur en fonction des spécifications vérifiées. Un piste de travail intéressante serait alors de choisir, parmi plusieurs mécanismes de fusion, le plus adapté à ces spécifications. Ce mécanisme de vérifications pourrait également être appliqué au modèle produit en sortie du tisseur. Il s'agirait alors de vérifier si l'application produite respecte un ensemble de propriétés souhaitées. Par exemple, il pourrait s'agir de contraintes de sécurité.

Dans le cadre de l'architecture sur 4 niveaux, la tâche de déployer les bonnes spécifications dans une situation donnée reviendrait alors au niveau tactique. Les résultats des vérifications réalisées sur le modèle produit après tissage pourraient alors être remontés au niveau tactique et pourrait entraîner le redéploiement de nouveaux réflexes externes. Ce redéploiement concernerait aussi bien les cascades

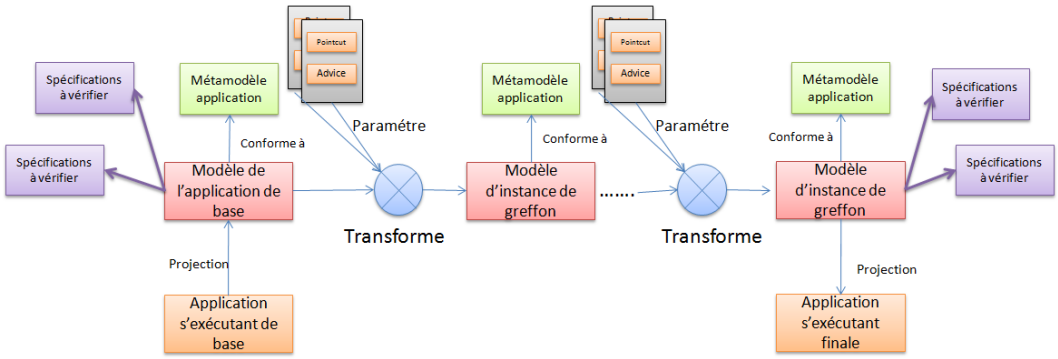


FIGURE 6.2 – Model checking

d'aspects que les spécifications.

6.2.3 Un méta-modèle d'assemblage et un langage de greffon applicable au plus grand nombre de plateformes d'exécution

Il pourrait également être intéressant de travailler plus en détail le méta-modèle d'application utilisé par les Aspects d'Assemblage afin que celui-ci soit applicable au plus grand nombre de plateformes d'exécution. Pour cela, plusieurs approches sont envisageables. La première consisterait à étudier les méta-modèles des plateformes existantes afin d'en faire une union tandis que la seconde consisterait à identifier l'ensemble des éléments nécessaire et suffisant devant se trouver dans le méta-modèle. Cela nécessitera une étude approfondie des plateformes mais aussi des ADL (Architecture Definition Langage) existant dans la littérature. Une étude similaire pourrait être réalisée au niveau du méta-modèle des greffons des AAs ainsi que de leur langage. L'étude pourrait permettre d'améliorer l'expressivité des AAs ainsi que leur lisibilité. Ces diverses études pourraient alors mener à augmenter les capacités des AAs, comme par exemple modifier le fait qu'une liaison est *obligatoire* ou *optionnelle*. Il conviendrait alors d'étudier dans quelle mesure de telles modifications n'entrainerait pas la perte de la propriété de symétrie de l'opération de tissage.

6.2.4 Vers un mécanisme de fusion plus évolué

Nous avons vu que le mécanisme de fusion proposé dans les AAs basé sur ISL4WComp repose sur un ensemble d'opérateurs dont la composition deux à deux à été prouvée symétrique. Il serait intéressant, dans des travaux futurs, d'améliorer ce mécanisme de fusion. Un premier axe de travail consisterait à augmenter le nombre de ces opérateurs et par conséquent de règles de composition qui leurs sont associées. Pour faciliter cette tâche, il serait profitable de disposer d'un mécanisme permettant de vérifier si les règles associées aux opérateurs peuvent bien être composées avec les autres règles de manière symétrique.

Une autre piste de travail consisterait à introduire des notions de sémantique dans ce mécanisme de fusion. Pour cela, un premier axe pourrait être de sélectionner l'opérateur de fusion par défaut entre deux règles (pour rappel, lorsque deux règles sont en conflits et n'utilisent pas d'opérateur, un opérateur de parallélisme est introduit entre elles), en fonction d'informations sémantiques associées à ces deux règles. Ensuite, cela pourrait consister à ajouter des contraintes sémantiques dans les règles

de fusion. Ce point soulève également la question d'intégrer de la sémantique dans les greffons des aspects.

6.2.5 Et si on arrête plus les cascades ?

Une autre piste de travail reposerait sur l'idée de « laisser couler les cascades », c'est-à-dire d'enchaîner les cycles de tissage de manière circulaire tant qu'il est possible d'adapter (FIGURE 6.3). Dans le processus de tissage d'une cascade, une fois le dernier cycle de tissage réalisé, cela consisterait à réaliser à nouveau le cycle de tissage 0 et ainsi de suite jusqu'à ce qu'aucun aspect ne puisse plus être tissé (la propriété d'idempotence garantissant que les mêmes règles ne sont pas dupliquées). Cela permettrait, par exemple, à des aspects d'un cycle de tissage de bénéficier des éléments instanciés par d'autres aspects de tous les cycles y compris les suivants et le sien. En particulier, cela permettrait à un aspect d'être déployé avec d'autres aspects dans un même cycle sans contraintes d'ordre, mais d'être tissés selon un ordre dirigé par l'opportunisme pour qu'un maximum d'aspects puissent être tissés. La fonctionnalité de perception par exemple pourrait elle-même s'améliorer.

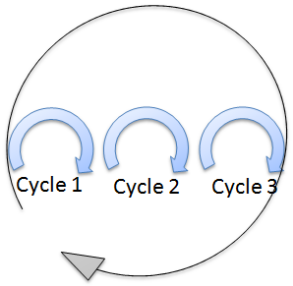


FIGURE 6.3 – Laisser coules les cascades

Un tel système offrirait alors de nombreuses similitudes avec les systèmes de réécriture, la question qui se poserait alors serait de savoir si le mécanisme d'adaptation est convergent. Cette propriété de convergence se définie elle-même comme en ensemble de deux sous-propriétés : la *confluence* et la *terminaison*. Si la propriété de symétrie de l'opération de tissage peut laisser supposer que la propriété de confluence est vérifiée, il apparaît clairement que la propriété de terminaison de l'est pas. En effet, si nous considérons qu'un aspect A, pour un cycle de tissage, ajoute un point de jonction permettant le tissage d'un autre aspect B, d'un autre cycle, et inversement que B ajoute un point de jonction permettant le tissage de A ; alors, l'opération de tissage n'est plus terminale. Cependant, la détection d'un tel phénomène semble pouvoir être identifiée et neutralisée lors du déploiement d'une cascade en représentant les interactions entre aspects sous la forme d'un graphe et en identifiant les cycles dans ce graphe.

6.2.6 Diagramme de feature et cascades d'aspects

Nous avons vu que les cascades d'aspects, en terme de variabilité, offrent des possibilités qui peuvent être exprimées sous la forme de diagramme de feature. En particulier, nous avons vu qu'elles permettent d'exprimer deux types de relations : « et » et « ou ». Dans les diagrammes de feature, d'autres opérateurs existent, il serait alors intéressant de faire évoluer les cascades afin qu'elles permettent d'en exprimer tous les types. En particulier, il serait alors possible, à partir d'un diagramme de features de construire des cascades dans une approche « top-down » et inversement pour une approche

« bottom-up ». Enfin, grâce aux capacités d'extensibilité offertes par les cascades et la propriété de symétrie, serait-il envisageable d'utiliser une telle approche pour réaliser des compositions de diagrammes de features à l'exécution ?

6.2.7 Faire cohabiter plusieurs architectures sur quatre niveaux

Nous avons vu que l'architecture sur 4 niveaux a pour objectif de gérer un assemblage de composant permettant d'orchestrer localement des dispositifs et par conséquent des services pour dispositifs. Cette architecture peut être vue comme globale lorsqu'elle orchestre l'ensemble des dispositifs du système. Cependant, il est possible d'envisager que les nœuds d'exécution sur lesquels sont déployés ces architectures sont mobiles. Il n'est donc pas possible d'anticiper leur présence. Il peut alors être envisagées d'utiliser l'architecture de manière décentralisée. Il serait ainsi possible d'avoir plusieurs nœuds d'exécution pour plusieurs architectures sur quatre niveaux qui pourraient réaliser des orchestrations locales autonomes et pourraient communiquer à travers elles de manière globale. La question de l'auto-organisation d'un tel système se pose ainsi que celle de la définition des interactions entre les différents systèmes. Les quatre niveaux de l'architecture sont-ils nécessaires ? A travers quels niveaux peuvent se faire les interactions ? Est-il envisageable d'obtenir une organisation hiérarchique ? Est-ce que l'architecture sur quatre niveaux peut être vue comme une architecture pour des agents ambiants ?

Comme nous pouvons le constater, de nombreuses pistes, à plus ou moins long terme, peuvent venir compléter les travaux réalisés dans cette thèse et ouvrent des portes vers des extensions ou collaborations dans de nombreux domaines.

6.3 Liste des publications

6.3.0.1 Chapitre de livre

Nicolas Ferry, Vincent Hourdin, Stéphane Lavirotte, Gaëtan Rey, Michel Riveill, Jean-Yves Tigli. "WComp, a Middleware for Ubiquitous Computing" in Ubiquitous Computing, chapter 8, pages 151–176, InTech, feb 2011, 978-953-307-409-2

6.3.0.2 Revues internationales

Nicolas Ferry, Stéphane Lavirotte, Jean-Yves Tigli, Gaëtan Rey, Michel Riveill. "Aspects of Assembly and Cascaded Aspects of Assembly : Logical and Temporal Properties" in International Journal of Computer Sciences Issues (IJCSI). Vol. 8. Issue 4. pages 1-15, August 2011 (acceptance rate = 29.7%).

6.3.0.3 Conférences internationales

Nicolas Ferry, Stéphane Lavirotte, Jean-Yves Tigli, Gaëtan Rey, Michel Riveill. "Multi-dynamics adaptations using Cascaded Aspect of Assembly" in Proceedings of the 9th Workshop on Adaptive and Reflective Middleware (ARM'2010), ACM, Bangalore, India, 29 nov - 3 dec 2010

Nicolas Ferry, Stéphane Lavirotte, Jean-Yves Tigli, Gaëtan Rey, Michel Riveill. "Toward a Behavioral Decomposition for Context-awareness and Continuity of Services" in Proceedings of the International Symposium on Ambient Intelligence (ISAmI), Guimarâes, Portugal, 16-18 june 2010 (Acceptation Rate : 40 %)

Nicolas Ferry, Stéphane Lavirotte, Jean-Yves Tigli, Gaëtan Rey, Michel Riveill. "Context Adaptative Systems based on Horizontal Architecture for Ubiquitous Computing" in Proceedings of the International Conference on Mobile Technology, Applications and Systems (Mobility), ACM, Nice, France, 2-4 september 2009 (Acceptation Rate : 35 %)

Nicolas Ferry, Vincent Hourdin, Stéphane Lavirotte, Gaëtan Rey, Jean-Yves Tigli, Michel Riveill. "Models at Runtime : Service for Device Composition and Adaptation" in Proceedings of the 4th International Workshop Models@run.time at Models 2009 (MRT'09), Denver, Colorado, USA, 5 october 2009 (Acceptation Rate : 32 %)

6.3.0.4 Conférence Nationale

Nicolas Ferry, Stéphane Lavirotte, Jean-Yves Tigli, Gaëtan Rey, Michel Riveill. "Cascaded Aspects of Assembly for Ubiquitous Computing" in Proceedings of th 6th International Conference SETIT'11, ieee, Sousse, Tunisia, 12 may - 15 may 2011

6.3.0.5 Rapports de recherche et livrables

Gaëtan Rey, Jean-Yves Tigli, Stéphane Lavirotte, **Nicolas Ferry**, Sana Fathallah, Joëlle Coutaz, Emeric Fontaine, Fabrice Jouanot, Marie-Christine Rousset, Philippe Renevier, Anne-Marie Pinna-Déry, Vincent Hourdin. "Modélisation du contexte et Adaptation" Research Report ANR Continuum,

number D2.1-2.2, 1-57 pages, aug 2010

Mireille Blay-Fornarino, **Nicolas Ferry**, Sébastien Mosser, Stéphane Lavirotte, Jean-Yves Tigli. "Démonstrateur de l'application SEDUITE" Research Report RNTL FAROS, number F.4.4, 1-52 pages, sep 2009

Noël Plouzeau, **Nicolas Ferry**, Mireille Blay-Fornarino, Anne-Françoise Le Meur, Sébastien Mosser, Lionel Seinturier, Jean-Yves Tigli, Guillaume Waignier. "Guide pour l'ecriture des transformation pivot vers plates-formes" Research Report RNTL FAROS, number F-2.5, 1-45 pages, jul 2009

Nicolas Ferry, "Adaptation Dynamique d'Applications au Contexte en Informatique Ambiante" Master's Thesis Université de Nice - Sophia Antipolis, 44 pages, Nice, France, sep 2008.

CHAPITRE 7

Bibliographie détaillée par catégorie

7.1 Bibliographie programmation orientée Aspect

[1] Glossary from aosd.net community wiki.

[2] M. Aksit, A. Rensink, and T. Staijen. A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 39–50. ACM, 2009.

Ils proposent une approche pour identifier les interférences entre aspects, et en particulier sur les *shared joinpoints*. L'approche est indépendante du language des aspects. Elle consiste à simuler et à représenter les différents états au runtime d'un programme sous la forme d'un graphe puis d'identifier les interférences comportementales entre les aspects et en particulier en fonction de leur ordre d'exécution .

[3] S. Apel and D. Batory. When to use features and aspects ? : a case study. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 59–68. ACM, 2006.

[4] S. Apel, T. Leich, and G. Saake. Aspectual mixin layers : aspects and features in concert. In *Proceedings of the 28th international conference on Software engineering*, pages 122–131. ACM, 2006.

Ce papier pose clairement les avantages et limites des approches orientées features et aspects, et montre une manière de mixer les deux. Entre autre les critères relatifs aux points de coupe hétérogènes et homogènes sont clairement définis .

[5] J. Barreiros and A. Moreira. Aspect interaction management with meta-aspects and advice cardinality. In *Workshop Proceedings ADI07*, page 11. Citeseer, 2007.

[6] A. Charfi, T. Dinkelaker, and M. Mezini. A plug-in architecture for self-adaptive web service compositions. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 35–42. IEEE, 2009.

[7] A. Charfi and M. Mezini. Aspect-Oriented Web Service Composition with AO4BPEL. *Web Services : European Conference, ECOWS 2004, Erfurt, Germany, September 27-30, 2004 : Proceedings*, 2004.

Les deux principaux problèmes de BPEL sont la modularité et les adaptations dynamiques de l'orchestration définie. En BPEL les orchestrations sont prédéfinies et statiques, il n'est pas possible de modifier une orchestration dynamiquement au runtime. Ils considèrent que la programmation par aspects peut résoudre ces problèmes grâce au tisseur qui peut apporter la dynamicité. D'autre part, la programmation par aspects permet l'ajout de ce qui est défini comme de la modularité transverse, c'est a dire la séparation des services d'une application et de ses propriétés non fonctionnelles (sécurité, persistance ...) .

[8] R. Douence, D. Le Botlan, J. Noyé, and M. Südholt. Concurrent aspects. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 79–88. ACM, 2006.

[9] R. Douence and M. Sudholt. A model and a tool for Event-based Aspect-Oriented Programming (EAOP). *LMO'03*, 2002.

Une évolution des points de coupe en points de coupe sophistiqués. En E-AOP il s'agit de décrire les points d'intérêts pour l'application d'un aspect comme des moments d'exécution (des événements) de l'application (approche relativement linéaire). Il s'agit donc en particulier d'étendre la notion habituelle de point de coupe pour que les aspects prennent alors la forme : eventPattern => action. Dans les points de coupe, des patrons d'événements sont désormais à prendre en compte (plus un seul mais plusieurs). Un moniteur permet de sélectionner les aspects qui seront évalués. Il s'agit donc ici de prendre en compte le contexte interne, la trace d'exécution. Le tisseur est alors considéré comme un moniteur .

[10] Sana Fathallah, Stéphane Lavirotte, Jean-Yves Tigli, and Gaëtan Rey. Résolution des interférences entre les adaptations par transformations de graphes. In *29ème INFORSID*, , Lille, France, May 2011.

[11] P. Grace, B. Lagaisse, E. Truyen, and W. Joosen. A reflective framework for fine-grained adaptation of aspect-oriented compositions. In *Software Composition*, pages 215–230. Springer, 2008.

[12] P. Greenwood and L. Blair. A framework for policy driven auto-adaptive systems using dynamic framed aspects. *Transactions on Aspect-Oriented Software Development II*, pages 30–65, 2006.

Ils proposent une approche totalement dynamique permettant l'adaptation d'applications à l'aide d'aspects . Pour ce faire, ils utilisent des policies contenant des règles ECA. Dans un premier temps, des aspects de monitoring sont déployés puis les policies évaluées dans l'optique de déclencher des aspects d'action. Pour cela ils utilisent AspectWerkz, les aspects s'appliquent sur des objets de manière invasive contrairement aux AA. Le déclenchement des adaptations est décrit dans les weave conditions des policies .

[13] P. Greenwood, B. Lagaisse, F. Sanen, G. Coulson, A. Rashid, E. Truyen, and W. Joosen. Interactions in AO middleware. In *Workshop on ADI, ECOOP*. Citeseer, 2007.

Ce papier traite des interactions entre aspects. Il pose clairement les définitions de quelques types d'interactions .

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. *ECOOP 2001 - Object-Oriented Programming*, pages 327–354, 2001.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97-Object-Oriented Programming*, pages 220–242. Springer, 1997.

[16] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A disciplined approach to aspect composition. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 68–77. ACM, 2006.

[17] A. Mendhekar, G. Kiczales, and J. Lamping. RG : A case-study for aspect-oriented programming. Technical report, 1997.

Ce papier présente un cas d'étude pour l'utilisation de l'AOP. Il s'agit d'optimiser un mécanisme de traitement d'image. Les trois types d'optimisation sont intégrées dans l'application sous la forme d'aspects. On retrouve entre autre dans ce papier des resultats expérimentaux sur des évaluations en terme de LOC de l'approche AOP. Comparé à du code OO naif ou à du code OO optimisé à la main, il apparait que pour cette application l'utilisation de l'AOP permet d'obtenir les meilleurs résultats en terme de performance du programme (temps et optimisation mémoire avec le plus faible nombre de duplications d'images). Mais surtout, comparé à la version optimisé le nombre de ligne de code développé (incluant le tisseur) est très nettement plus faible. : de l'ordre de 88% de réduction .

[18] J. Munnelly, S. Fritsch, and S. Clarke. An aspect-oriented approach to the modularisation of context. In *Pervasive Computing and Communications, 2007. PerCom'07. Fifth Annual IEEE International Conference on*, pages 114–124. IEEE, 2007.

Munelly et al proposent de décomposer le contexte en différentes catégories et de réaliser les adaptations relatives à ces contextes à l'aide d'aspects. Les aspects sont appliqués au-dessus d'objets. Cette décomposition est intéressante et permet de considérer un contexte autre que l'infrastructure sous-jacente. Les interferences entre aspects ne sont pas gérées et les informations contextuelles sont des paramètres de l'adaptation. Les aspects sont déclenchés de manière classique et non pas en fonction du contexte .

[19] A. Navasa, M.A. Pérez, J.M. Murillo, and J. Hernández. Aspect oriented software architecture : a structural perspective. In *Workshop on Early Aspects, AOSD*, volume 2. Citeseer, 2002.

Ce papier positionne les avantages de l'AOP et plus particulièrement les avantages de l'AOD sur le design d'architectures. Enfin, il évoque un certain nombre de points que doivent respecter les approche pour faire de l'AOD architecturelle, c'est-à-dire pour travailler sur la structure. .

[20] J. Perez, I. Ramos, J. Jaen, P. Letelier, and E. Navarro. Prisma : towards quality, aspect oriented and dynamic software architectures. In *Quality Software, 2003. Proceedings. Third International Conference on*, pages 59–66. IEEE, 2003.

Ce papier présente Prisma, une plateforme adaptative couplant à la fois AOP et composants. Le but est d'avoir une adaptation qui conserve l'état du composant à modifier et minimise l'influence de cette adaptation sur les autres composants qui sont liés avec le composant adapté. Prisma définit deux modes d'adaptation logicielle : l'adaptation de l'architecture (ajouter supprimer des éléments architecturaux) et l'adaptation interne d'un composant et ou système (modifier un aspect, tissage ou port). Dans les deux cas l'adaptation est considérée comme une préoccupation qui est encapsulée dans des aspects. Dans PRISMA, un composant est décrit comme une agrégation d'aspects interchangeables dynamiquement .

[21] M. Pinto, L. Fuentes, and J.M. Troya. A dynamic component and aspect-oriented platform. *The Computer Journal*, 48(4) :401, 2005.

[22] F. Sanen, E. Truyen, and W. Joosen. Modeling context-dependent aspect interference using default logics. In *RAM-SE08-ECOOP08 Workshop on Reflection, AOP, and Meta-Data for Software Evolution*, page 15. Citeseer, 2008.

[23] F. Sanen, E. Truyen, W. Joosen, A. Jackson, A. Nedos, S. Clarke, N. Loughran, and A. Rashid. Classifying and documenting aspect interactions. *Aspects, Components, and Patterns for Infrastructure Software*, page 23, 2006.

[24] J. Whittle and P. Jayaraman. Mata : A tool for aspect-oriented modeling based on graph transformation. *Models in Software Engineering*, pages 16–27, 2008.

[25] Z. Yang, B.H.C. Cheng, R.E.K. Stirewalt, J. Sowell, SM Sadjadi, and P.K. McKinley. An aspect-oriented approach to dynamic adaptation. In *Proceedings of the first workshop on Self-healing systems*, pages 85–92. ACM, 2002.

[26] A. Zambrano, S. Gordillo, and I. Jaureguiberry. Aspect-based adaptation for ubiquitous software. *Mobile and Ubiquitous Information Access*, pages 136–140, 2004.

Ce papier explique comment l'AOP peut être utilisée pour la création d'applications ubiquitaires. L'AOP permet d'offrir un bonne modularisation et séparation des préoccupations transverse. Elle permettrait alors dans le cadre de l'informatique ambiante d'isoler des préoccupations variables de l'application. Les Aspects permettent alors d'isoler, réutiliser ou encore échanger ces préoccupations en fonction de l'environnement de l'application. La préoccupation de prise en compte du contexte peut alors être séparée du reste de l'application. De plus l'informatique ambiante met en jeu des systèmes dont les designers doivent considérer les caractéristiques suivantes : mobilité,

variabilité des ressources et de leur accessibilité. Ces variabilités ont pour conséquence dans un approche classique de forcer la réécriture de même préoccupations pour considérer ces variantes. L'AOP permet de réduire la réécriture de ces préoccupations grâce à l'abstraction offerte par les points de coupe .

7.2 Bibliographie programmation orientée feature

[27] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5) :49–84, 2009.

Présentation complète, détaillée et de référence du FOSD. Papier complet pour introduire au domaine. .

[28] D. Batory. Feature models, grammars, and propositional formulas. *Software Product Lines*, pages 7–20, 2005.

[29] D. Batory. Using modern mathematics as an fosd modeling language. In *Proceedings of the 7th international conference on Generative programming and component engineering*, pages 35–44. ACM, 2008.

[30] C.H.P. Kim, C. Kästner, and D. Batory. On the modularity of feature interactions. In *Proceedings of the 7th international conference on Generative programming and component engineering*, pages 23–34. ACM, 2008.

7.3 Bibliographie Contexte

[31] A. Benyelloul, F. Jouanot, and M.C. Rousset. Conquer : an rdfs-based model for context querying. In *UbiMob*, 2010.

[32] C. Bolchini, C.A. Curino, E. Quintarelli, F.A. Schreiber, and L. Tanca. A data-oriented survey of context models. *ACM SIGMOD Record*, 36(4) :19–26, 2007.

[33] T. Buchholz, A. Küpper, and M. Schiffers. Quality of context : What it is and why we need it. In *Proceedings of the workshop of the HP OpenView University Association*, pages 1–13. Citeseer, 2003.

[34] G. Chen and D. Kotz. A survey of context-aware mobile computing research. Technical report, 2000.

[35] G. Chen, D. Kotz, and DARTMOUTH COLL HANOVER NH DEPT OF COMPUTER SCIENCE. Context-sensitive resource discovery. In *Mobile Computing Systems and Applications, 2002. Proceedings Fourth IEEE Workshop on*, pages 94–104. Citeseer, 2003.

[36] J. Coutaz, J.L. Crowley, S. Dobson, and D. Garlan. Context is key. *Communications of the ACM*, 48(3) :49–53, 2005.

Dans ce papier, à travers un exemple complet, l'auteur nous expose l'importance de la notion de contexte et propose une pyramide modélisant les étapes nécessaires à une bonne prise en compte du contexte. Les couches composant la pyramide sont capture d'observables-transformation en observable symbolique, identification situation, exploitation. La justesse de celle-ci est vérifiée au travers un exemple qui fait preuve de concept.

[37] A.K. Dey. Understanding and using context. *Personal and ubiquitous computing*, 5(1) :4–7, 2001.

[38] K. Geihs, R. Reichle, M. Wagner, and M. Khan. Modeling of context-aware self-adaptive applications in ubiquitous and service-oriented environments. *Software Engineering for Self-Adaptive Systems*, pages 146–163, 2009.

[39] K. Henricksen, J. Indulska, and A. Rakotonirainy. Modeling context information in pervasive computing systems. *Pervasive Computing*, pages 79–117, 2002.

[40] H. Lei, D.M. Sow, II Davis, S. John, G. Banavar, and M.R. Ebling. The design and applications of a context service. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4) :45–55, 2002.

[41] J. Pauty, M. Banâtre, and P. Couderc. Logical versus physical programming for ubiquitous applications. In *First Workshop on Intelligent Solutions in Embedded Systems (WISES)*, volume 13. Citeseer, 2003.

Ce papier présente une approche de traitement du contexte dite « physique ». L'objectif est alors de prendre en compte les informations contextuelles accessibles par les capteurs se trouvant à proximité d'une entité de référence .

[42] T. Strang and C. Linnhoff-Popien. A context modeling survey. In *Workshop on Advanced Context Modelling, Reasoning and Management as part of UbiComp*, 2004.

Ce papier présente les différentes manières de modéliser le contexte (ontologies,UML...) avec leurs avantages et inconvénients. Parmi ces critères ont trouve les suivants : le modèle doit pouvoir supporter de nombreuses données et fournir des indications sur leurs qualités. De même le modèle doit pouvoir representer les ambiguités présentes ou les informations absentes dans les données contextuelles. Le niveau de formalisme du modèle ne doit pas être trop complexe pour son utilisateur. Il doit pouvoir être distribué facilement et si possible composé dynamiquement à partir de diverses sources.

[43] M. Strimpakou, I. Roussaki, C. Pils, M. Angermann, P. Robertson, and M. Anagnostou. Context modelling and management in ambient-aware pervasive environments. *Location-and Context-Awareness*, pages 2–15, 2005.

7.4 Bibliographie Informatique ambiante, vision et challenges

[44] G. Banavar and A. Bernstein. Software infrastructure and design challenges for ubiquitous computing applications. *Communications of the ACM*, 45(12) :92–96, 2002.

[45] G. Banavar and A. Bernstein. Challenges in design and software infrastructure for ubiquitous computing applications. *Advances in computers*, 62 :179–202, 2004.

[46] C.A. da Costa, A.C. Yamin, and C.F.R. Geyer. Toward a general software infrastructure for ubiquitous computing. *IEEE Pervasive Computing*, pages 64–73, 2008.

[47] James L. Crowley Joëlle Coutaz. Plan « intelligence ambiante » : défis et opportunités. Technical report, département ST2I du CNRS et du Groupe de Travail « Intelligence Ambiante » du Groupe de Concertation Sectoriel (GCS3), 2011.

[48] T. Kindberg and A. Fox. System software for ubiquitous computing. *Pervasive Computing, IEEE*, 1(1) :70–81, 2002.

[49] S. Preuß and C.H. Cap. Overview of spontaneous networking-evolving concepts and technologies. *Rostocker Informatik-Berichte*, 24 :113–123, 1999.

[50] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 1–7. ACM New York, NY, USA, 1996.

Dans ce papier, ce qui nous intéresse est la taxonomie des stratégies d'adaptation. L'auteur décrit un continuum entre deux stratégies extremes qui sont les adaptations "laissez faire" gérées par l'application et les adaptation

"transparentes" gérées par le système (attention cette notion de gérée par le système implique chez lui une certaine centralisation pour, par exemple, de la résolution de conflits) .

[51] M. Satyanarayanan. Pervasive computing : Vision and challenges. *IEEE Personal communications*, 8(4) :10–17, 2001.

[52] M. Weiser. The Computer for the Twenty-First Century. *Scientific American*, 265(3) :94–104, 1991. Dans ce papier, Weiser expose sa vision de l'avenir de l'ordinateur et pose la notion d'informatique ambiante. Il parle alors d'une informatique invisible, présente en tout lieu en tout instant et en toute chose. Il oppose cette approche à la réalité virtuelle, qui consiste à recréer le monde dans une machine et non pas à avoir des machines dans le monde. Apparait alors l'idée que, dans ce cadre de l'informatique ubiquitaire, il va falloir adapter le comportement des machines . Il imagine donc des espaces avec une multiplicité de dispositifs intelligents et mobiles dont l'accumulation fournirait une puissance de calcul conséquente. L'informatique ubiquitaire ne s'impose alors plus comme une barrière pour les interactions humaines .

7.5 Bibliographie Intergiciels

[53] D. Athanasopoulos, A.V. Zarras, V. Issarny, E. Pitoura, and P. Vassiliadis. Cowsami : Interface-aware context gathering in ambient intelligence environments. *Pervasive and Mobile Computing*, 4(3) :360–389, 2008.

[54] N. Bencomo. On the use of software models during software execution. In *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering*, pages 62–67. IEEE Computer Society, 2009.

[55] N. Bencomo, P. Grace, C. Flores, D. Hughes, and G. Blair. Genie : Supporting the model driven development of reflective, component-based adaptive systems. In *Proceedings of the 30th international conference on Software engineering*, pages 811–814. ACM, 2008.

[56] H. Blohm and J.J. Dubray. Service Component Architecture. 2005.

[57] A. Bottaro, J. Bourcier, C. Escoffier, and P. Lalanda. Context-Aware Service Composition in a Home Control Gateway. In *IEEE International Conference on Pervasive Services*, pages 223–231, 2007.

Ce papier présente un mécanisme de composition de services contextuels. Ce mécanisme repose sur iPOJO qui offre des mécanismes pour la composition dynamique de service au-dessus d'OSGi. Le but de cette architecture est de permettre de créer des applications rééllement dynamiques qui peuvent intergrer de nouvelles conditions au runtime. Cette architecture est dédiée au home control et reste centralisée (dans le context service par exemple). Le mécanisme de décision permet en fonction du contexte de binder les services entre eux de la manière la plus adaptée à la situation. Pour ce faire, ils utilisent le local autonomic manager. Dans ce papier on trouve en introduction une bonne mise en avant des avantages de la programmation orienté service mais aussi des requis pour les applications dans les batiments intelligents .

[58] A. Bouzeghoub, C. Taconet, A. Jarraya, N.K. Do, and D. Conan. Complementarity of process-oriented and ontology-based context managers to identify situations. In *Digital Information Management (ICDIM), 2010 Fifth International Conference on*, pages 222–229. IEEE, 2010.

[59] J. Branke, M. Mnif, C. Muller-Schloer, and H. Prothmann. Organic computing–addressing complexity by controlled self-organization. In *Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on*, pages 185–191. IEEE, 2006.

[60] J. Bronsted, K.M. Hansen, and M. Ingstrup. A survey of service composition mechanisms in ubiquitous computing. In *Proceedings of UbiComp 2007 Workshop*, volume 4717, pages 87–92. Citeseer, 2007.

Ce papier propose un état de l'art des techniques de composition de services proposées dans différents projets ou middleware pour l'informatique ambiante. Il met en avant que cette capacité est particulièrement importante en informatique ambiante . Parmi les critères utilisés dans cette étude on retrouve : quand sont spécifiées les composition, si les approches ont plutôt un déploiement centralisé ou décentralisé ou encore si l'infrastructure sur lesquels ils travaillent peut être variable et imprévisible. Les divers résultats présentés dans le tableau ne sont pas toujours justifiés. (Remarque les colonnes du tableau sont décalées d'une case ...) .

[61] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.B. Stefani. The fractal component model and its support in java. *Software : Practice and Experience*, 36(11-12) :1257–1284, 2006.

[62] L. Capra. *Reflective Mobile Middleware for Context-Aware Applications*. PhD thesis, University of London, 2003.

Dans cette thèse sont présentés les travaux concernant l'intergiciel CARISMA qui a pour but de fournir des mécanismes permettant la prise en compte des changements de contexte en définissant des politiques de réaction. Les règles définissant ces politiques peuvent être conflictuelles. Le middleware a également pour rôle de résoudre ces conflits. Dans l'optique de permettre au middleware de modifier son comportement en fonction du contexte d'exécution, CARISMA utilise la réflexivité associée à des métadonnées. .

[63] C. Cetina, P. Giner, J. Fons, and V. Pelechano. Autonomic computing through reuse of variability models at runtime : The case of smart homes. *Computer*, 42(10) :37–43, 2009.

[64] Daniel CHEUNG-FOO-WO. *Adaptation dynamique par tissage d'aspects*. PhD thesis, l'université de Nice-Sophia Antipolis, 2009.

[65] M. Clarke, G.S. Blair, G. Coulson, and N. Parlavantzas. An efficient component model for the construction of adaptive middleware. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 160–178. Springer-Verlag, 2001.

[66] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Transactions on Computer Systems (TOCS)*, 26(1) :1–42, 2008.

[67] M. Dalmau, P. Roose, and S. Laplace. Context aware adaptable applications-a global approach. *International Journal of Computer Science*, 1(2009) :12, 2009.

[68] P.C. David. *Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation*. PhD thesis, Université de Nantes, 2005.

Cette thèse présente le middleware SAFRAN qui se base au-dessus de Fractal. Il utilise une approche par aspects pour modulariser le code d'adaptation d'applications, mais aussi afin d'utiliser la dynamicité offerte par le tissage dynamique. Cet intergiciel tend donc à faciliter le développement d'applications sensibles au contexte mais ne réalise pas d'interprétation de ce contexte afin d'en déduire des informations de plus haut niveau.

[69] P.C. David and T. Ledoux. Wildcat : a generic framework for context-aware applications. In *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–7. ACM, 2005.

[70] P.C. David and T. Ledoux. WildCAT : a generic framework for context-aware applications. *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–7, 2005.

[71] P.C. David and T. Ledoux. Une approche par aspects pour le développement de composants Fractal adaptatifs. *RSTI-Série L'Objet (RSTI-Objet)*, 12(2-3), 2006.

[72] P.C. David, T. Ledoux, M. Léger, and T. Coupaye. Fpath and fscript : Language support for navigation and reliable reconfiguration of fractal architectures. *Annals of Telecommunications*, 64(1) :45–63, 2009.

[73] A.K. Dey, D. Salber, M. Futakawa, and G. Abowd. An architecture to support context-aware applications. *submitted to UIST*, 99.

Ce papier présente la context toolkit, qui est l'infrastructure de référence du domaine de la capture du contexte. A la manière des « GUI widgets » que l'on trouve dans les boites à outils graphiques et qui sont des composants de base réutilisables liant applications et utilisateurs ; la context toolkit offre des « context widgets » qui font la liaison entre les applications et leurs environnement. A ce niveau là, il s'agit d'un intergiciel uniquement centré sur la capture d'observables et la déduction d'informations à partir de ces derniers puisque les enactors n'ont pas encore étés introduits.

[74] Nicolas Ferry, Vincent Hourdin, Stéphane Lavirotte, Gaëtan Rey, Michel Riveill, and Jean-Yves Tigli. *WComp, a Middleware for Ubiquitous Computing*, chapter 8, pages 151–176. InTech, February 2011.

[75] Nicolas Ferry, Stéphane Lavirotte, Jean-Yves Tigli, Gaëtan Rey, and Michel Riveill. Multi-dynamics adaptations using Cascaded Aspect of Assembly. In *11th International Middleware Conference (Workshop on Adaptive and Reflective Middleware)(ARM'2010), poster*, Bangalore, India, December 2010. ACM.

[76] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, E. Gjorven, S. ICT, and N. Trondheim. Using architecture models for runtime adaptability. *IEEE software*, 23(2) :62–70, 2006.

Ce papier présente l'intergiciel MADAM, ce dernier propose de réaliser des adaptations en modifiant l'architecture d'une application (à base de composants). Il est important de noter que cet intergiciel repose sur une architecture centralisée et les auteurs reconnaisse que le passage à l'echelle de ce middleware reste un point difficile surtout du point de vue de la réactivité. .

[77] J. Fox and S. Clarke. Exploring approaches to dynamic adaptation. In *Proceedings of the 3rd International DiscCoTec Workshop on Middleware-Application Interaction*, pages 19–24. ACM, 2009.

[78] Paul Naoumenko Daniele Miorandi Janne Lahti Helena Rivas Daniel Schreckling Francoise Baude, Ludovic Henrio. Evaluating the fitness of service compositions. Technical Report WP3.2, BIONETS Deliverable 3.2.5, February 2009.

[79] Emeric Fontaine Teresa Colombi Aurore Russo Léonid Synyukov Stéphane Lavirotte Gaëtan Rey Jean Yves Tigli Gaëlle Calvary, Joëlle Coutaz. Méta-IHM et points de contrôle utilisateur dans CONTINUUM. Technical Report D4.1-4.2, ANR Continuum, September 2010.

[80] D. Garlan, S.W. Cheng, A.C. Huang, B. Schmerl, and P. Steenkiste. Rainbow : Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10) :46–54, 2004.

[81] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, pages 27–32. ACM, 2002.

[82] Paul Grace, Geoff Coulson, Gordon Blair, Barry Porter, and Danny Hughes. Dynamic reconfiguration in sensor middleware. In *Proceedings of the international workshop on Middleware for sensor networks*, MidSens '06, pages 1–6, New York, NY, USA, 2006. ACM.

[83] T. Gu, H.K. Pung, and D. Zhang. Peer-to-peer context reasoning in pervasive computing environments. In *Pervasive Computing and Communications, 2008. PerCom 2008. Sixth Annual IEEE International Conference on*, pages 406–411. IEEE, 2008.

[84] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *Computer*, 41(4) :93–95, 2008.

[85] Viet Dung Doan Paul Naoumenko Daniele Miorandi David Lowe Mihaela Ion Lahti Janne Heiko Pfeffer, Louay Bassbouss. Bio inspired service creation and evolution. Technical report, BIONETS Deliverable 3.2.6, July 2010.

[86] Vincent Hourdin. *Contexte et Sécurité dans les Intergiciels d'Informatique Ambiante*. PhD thesis, Université de Nice - Sophia Antipolis, July 2010.

[87] Vincent Hourdin, Jean-Yves Tigli, Stéphane Lavirotte, Gaëtan Rey, and Michel Riveill. SLCA, composite services for ubiquitous computing. In *Proc. of the 5th Int. Conf. on Mobile Technology, Applications and Systems (Mobility)*, September 2008.

[88] P. Huang, V. Lenders, P. Minnig, and M. Widmer. Mini : A minimal platform comparable to Jini for ubiquitous computing. In *International Symposium on Distributed Objects and Applications (DOA), Irvine*, 2002.

[89] N. Ibrahim, F. Le Mouël, and S. Frénot. Semantic service substitution in pervasive environments. *International Journal of Services, Economics and Management (IJSEM)*, 2011. "Service-Oriented Engineering" special issue.

[90] J.Y. Jung, J. Park, S.K. Han, and K. Lee. An eca-based framework for decentralized coordination of ubiquitous web services. *Information and Software Technology*, 49(11-12) :1141–1161, 2007.

[91] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1) :41–50, 2003.

[92] Xavier Blanc Anthony Cleve Laurence Duchien, Carlos Parra. Lignes de production de logiciels pour applications mobiles. Ecole Thématique Intelligence Ambiante (ETIA 2011), 2011.

[93] D. Marples and P. Kriens. The open services gateway initiative : An introductory overview. *Communications Magazine, IEEE*, 39(12) :110–114, 2001.

[94] C. Mascolo, L. Capra, and W. Emmerich. Middleware for mobile computing (a survey). In *Advanced Lectures on Networking, Lecture Notes in Computer Science*, volume 2497, pages 20–58, 2002.

[95] P.K. McKinley, S.M. Sadjadi, E.P. Kasten, and B.H.C. Cheng. A Taxonomy of Compositional Adaptation. *Rapport Technique numéroMSU-CSE-04-17, juillet*, 2004.

Il s'agit d'une version moins détaillée du papier de Sadjadi. .

[96] B. Morin. *Leveraging Models from Design-time to Runtime to Support Dynamic Variability*. PhD thesis, Université de Rennes 1, 2010.

[97] B. Morin, O. Barais, G. Nain, and J.M. Jezequel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering*, pages 122–132. IEEE Computer Society, 2009.

[98] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, and Arnor Solberg. Models at runtime to support dynamic adaptation. *IEEE COMPUTER*, pages 46–53, 2009.

[99] C. Parra. *Towards Dynamic Software Product Lines : Unifying Design and Runtime Adaptations*. PhD thesis, Université des Sciences et Technologies de Lille, 2011.

[100] N. Paspallis, R. Rouvoy, P. Barone, G. Papadopoulos, F. Eliassen, and A. Mamelli. A pluggable and reconfigurable architecture for a context-aware enabling middleware system. *On the Move to Meaningful Internet Systems : OTM 2008*, pages 553–570, 2008.

[101] N. Pessemier, L. Seinturier, T. Coupaye, and L. Duchien. A model for developing component-based and aspect-oriented systems. In *Software Composition*, pages 259–274. Springer, 2006.

[102] Heiko Pfeffer, Steffen Krüssel, and Stephan Steglich. A fuzzy logic based model for representing and evaluating service composition properties. In *The Third International Conference on Systems and Networks Communications*, volume 0, pages 335–342, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[103] G.P. Picco, A.L. Murphy, and G.C. Roman. Lime : Linda meets mobility. In *Proceedings of the 21st international conference on Software engineering*, pages 368–377. ACM, 1999.

[104] A. Rasche and A. Polze. Configuration and dynamic reconfiguration of component-based applications with microsoft. net. 2003.

[105] Gaëtan Rey, Jean-Yves Tigli, Stéphane Lavirotte, Nicolas Ferry, Sana Fathallah, Joëlle Coutaz, Emeric Fontaine, Fabrice Jouanot, Marie-Christine Rousset, Philippe Renevier, Anne-Marie Pinna-Déry, and Vincent Hourdin. Modélisation du contexte et Adaptation. Technical Report D2.1-2.2, ANR Continuum, August 2010.

[106] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R.H. Campbell, and K. Nahrstedt. Gaia : a middleware platform for active spaces. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4) :65–67, 2002.

.

[107] Daniel Romero, Romain Rouvoy, Lionel Seinturier, Sophie Chabridon, Denis Conan, and Nicolas Pessemier. Enabling Context-Aware Web Services : A Middleware Approach for Ubiquitous Environments. In Michael Sheng, Jian Yu, and Schahram Dustdar, editors, *Enabling Context-Aware Web Services : Methods, Architectures, and Technologies*, pages 113–135. Chapman and Hall/CRC, May 2010.

[108] R. Rouvoy, M. Beauvois, and F. Eliassen. Dynamic aspect weaving using a planning-based adaptation middleware. In *Proceedings of the 2nd workshop on Middleware-application interaction : affiliated with the DisCoTec federated conferences 2008*, pages 31–36. ACM, 2008.

[109] R. Rouvoy, F. Eliassen, J. Floch, S. Hallsteinsen, and E. Stav. Composing components and services using a planning-based adaptation middleware. In *Software Composition*, pages 52–67. Springer, 2008.

[110] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *Mobile Computing Systems and Applications, 1994. WMCSA'08.*, pages 85–90, 1994.

Ce papier présente de vieux travaux sur ParcTab. Ils permettent de mettre en avant trois catégories d'informations contextuelles : domaine utilisateur (des profils), domaine système (infrastructure logicielle), domaine environnemental (tout ce qui entoure). .

[111] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *6th IEEE International Conference on Service Computing (SCC'09)*, pages 268–275, Bangalore, Inde, 2009. IEEE. CORE A. Acceptance rate : 18% (35/189).

[112] D. Sykes, W. Heaven, J. Magee, and J. Kramer. From goals to components : a combined approach to self-management. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 1–8. ACM, 2008.

[113] Jean-Yves Tigli, Stéphane Lavirotte, Gaëtan Rey, Vincent Hourdin, and Michel Riveill. Lightweight Service Oriented Architecture for Pervasive Computing. *International Journal of Computer Science Issues (IJCSI)*, 4, September 2009.

[114] Jean-Yves Tigli, Stéphane Lavirotte, Gaëtan Rey, Vincent Hourdin, Daniel Cheung, Eric Callegari, and Michel Riveill. WComp Middleware for Ubiquitous Computing : Aspects and Composite Event-based Web Services. *Annals of Telecommunications (AoT)*, 64, 2009.

[115] D. Weyns, S. Malek, and J. Andersson. On decentralized self-adaptation : Lessons from the trenches and challenges for the future. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 84–93. ACM, 2010.

[116] S.S. Yau and F. Karim. An adaptive middleware for context-sensitive communications for real-time applications in ubiquitous computing environments. *Real-Time Systems*, 26(1) :29–61, 2004.

[117] C. Zhang and H.A. Jacobsen. Resolving feature convolution in middleware systems. *ACM SIGPLAN Notices*, 39(10) :188–205, 2004.

7.6 Bibliographie Robotique

[118] R.A. Brooks. Elephants Don't Play Chess. *Designing Autonomous Agents : Theory and Practice from Biology to Engineering and Back*, pages 3–15, 1991.

Dans ce papier de robotique et d'IA, l'auteur parle de la notion d'activité situé. Il explique qu'il s'agit de définir une nouvelle méthodologie de construction de systèmes intelligents qui se basent sur l'infrastructure du système et en une décomposition de l'intelligence en différents modules de base qui définissent un comportement. Améliorer l'intelligence revient alors à ajouter un nouveau module. L'auteur exprime aussi l'idée que le monde est lui-même son meilleur modèle et par conséquent qu'en possèder un modèle perd de son sens. Il présente ensuite sur architecture de subsumption basée sur une décomposition comportementale.

[119] J.J. Bryson. *Intelligence by design : Principles of modularity and coordination for engineering complex adaptive agents*. PhD thesis, 2001.

Cette thèse présente en détail et clairement la décomposition comportementale .

[120] E. Gat et al. On three-layer architectures. *Artificial intelligence and mobile robots*, pages 195–210, 1997.

Ce papier présente une architecture sur trois niveaux pour les robots. Cette architecture propose de répondre aux limitations des architectures comportementales des Brooks .

[121] W. Heaven, D. Sykes, J. Magee, and J. Kramer. A case study in goal-driven architectural adaptation. *Software Engineering for Self-Adaptive Systems*, pages 109–127, 2009.

Ce papier propose d'utiliser les architectures 3-layer de la robotique pour les systèmes auto-adaptatifs. Le papier propose d'utiliser un case study pour montrer que cela est bien réalisable .

[122] J. Kramer and J. Magee. Self-managed systems : an architectural challenge. In *2007 Future of Software Engineering*, pages 259–268. IEEE Computer Society, 2007.

[123] PN Stamatis, ID Zaharakis, and AD Kameas. Exploiting ambient information into reactive agent architectures. In *Intelligent Environments. IE 06. 2nd IET International Conference on*, volume 2, pages 69–79. IET, 2006.

7.7 Bibliographie Modèles

[124] J. Bézivin and O. Gerbé. Towards a precise definition of the omg/mda framework. In *ase*, page 273. Published by the IEEE Computer Society, 2001.

[125] G. Blair, N. Bencomo, and R.B. France. Models@ run. time. *Computer*, 42(10) :22–27, 2009.

[126] T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152 :125–142, 2006.

Une taxonomie des transformations de modèles. Il permet de bien positionner toutes les catégories de transformations. .

[127] Sébastien Mosser. *Behavioral Compositions in Service-Oriented Architecture*. PhD thesis, Université Nice - Sophia Antipolis, ED STIC, Nice, France, October 2010.

[128] P.A. Muller, F. Fleurey, and J.M. Jézéquel. Weaving executability into object-oriented meta-languages. *Model Driven Engineering Languages and Systems*, pages 264–278, 2005.

[129] Pierre-Alain Muller, Frédéric Fondement, and Benoit Baudry. Modeling Modeling. In *ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, pages 2–16, Denver, Colorado, USA, Etats-Unis, 2009.

[130] Noël Plouzeau, Nicolas Ferry, Mireille Blay-Fornarino, Anne-Françoise Le Meur, Sébastien Mosser, Lionel Seinturier, Jean-Yves Tigli, and Guillaume Waignier. Guide pour l'ecriture des transformation pivot vers plates-formes. Research Report F-2.5, RNTL FAROS, July 2009.

[131] M. Svahnberg, J. Van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Software : Practice and Experience*, 35(8) :705–754, 2005.

[132] T. Vogel, A. Seibel, and H. Giese. Toward Megamodels at Runtime. In *Proceedings of the 5th International MODELS Workshop on Models@ run. time*, volume 641, pages 13–24, 2010.

Dans ce papier, nos travaux, présentés à models@run.time 09 sont présentés comme une approche permettant de tisser des variantes de systèmes (ou des configurations) dans des systèmes, à un niveau architectural. .

7.8 Bibliographie Autres

[133] K. Arnold, R. Scheifler, J. Waldo, B. O'Sullivan, and A. Wollrath. *Jini Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[134] Laurent Berger. *Mise en Œuvre des Interactions en Environnements Distribués, Compilés et Fortement Typés : le Modèle MICADO*. Phd thesis, University of Nice-Sophia Antipolis, oct 2001.

[135] J. Buisson. *Adaptation dynamique de programmes et composants parallèles*. Thèse de Doctorat, Institut National des Sciences Appliquées de Rennes, 2006.

[136] H. Chen, D. Chakraborty, L. Xu, A. Joshi, and T. Finin. Service discovery in the future electronic market. In *Proc. Workshop on Knowledge Based Electronic Markets, AAAI2000, Austin*, 2000.

[137] P.C. Clements. A survey of architecture description languages. In *Proceedings of the 8th international workshop on software specification and design*, page 16. IEEE Computer Society, 1996.

[138] J.L. Crowley, J. Coutaz, and F. Bérard. Perceptual user interfaces : things that see. *Communications of the ACM*, 43(3), 2000.

[139] P.T. Eugster, P.A. Felber, R. Guerraoui, and A.M. Kermarrec. The many faces of publish/subscribe. *ACM computing Surveys*, 35(2) :114–131, 2003.

[140] Erik Guttman. Service Location Protocol : Automatic Discovery of IP Network Services. *IEEE Internet Computing*, 3 :71–80, 1999.

[141] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4) :287–317, 1983.

[142] Walter L. Hürsch and Cristina Videira Lopes. Separation of concerns. Technical report, 1995.

[143] G. Kiczales, J.M. Ashley, L. Rodriguez, A. Vahdat, and D.G. Bobrow. Metaobject protocols : Why we want them and what else they can do. *Object-Oriented Programming : The CLOS Perspective*, pages 101–118, 1993.

[144] J. Kramer and J. Magee. The evolving philosophers problem : Dynamic change management. *IEEE Transactions on software engineering*, pages 1293–1306, 1990.

[145] T. Ledoux, M. Blay, E. Bruneton, D. Caromel, T. Coupaye, D. Hagimont, J.M. Menaud, J. Noyé, and M. Riveill. Etat de l'art sur l'adaptabilité. Technical report, Livrable D, 2001.

[146] I.S. MacKenzie and C. Ware. Lag as a determinant of human performance in interactive systems. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, pages 488–493. ACM New York, NY, USA, 1993.

Ce papier présente une étude sur l'impact du lag dans les systèmes interactifs sur les performances utilisateurs. Après quelques explications sur la définition du lag et son mode de calcul, des résultats d'expérimentation nous sont proposés. Ainsi avec un lag de 75ms les effets se font vraiment ressentir sur l'utilisation du système, par contre entre 8,5ms et 25ms il n'y a pas vraiment d'impact .

[147] Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter Brown, and Rebekah Metz. Reference model for service oriented architecture 1.0. Technical report, OASIS, 2006.

Il s'agit du papier de référence pour la présentation des SOA 1.0 par OASIS. .

[148] P. Maes. Concepts and experiments in computational reflection. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 147–155. ACM, 1987.

[149] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering, IEEE Transactions on*, 26(1) :70–93, 2000.

[150] C. Müller-Schloer. Organic computing : on the feasibility of controlled emergence. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 2–5. ACM, 2004.

[151] A. Newberger and A. Dey. Designer support for context monitoring and control. *IRB-TR-03-017, Intel Research Berkeley*, 2003.

[152] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf. An architecture-based approach to self-adaptive software. *Intelligent Systems and Their Applications, IEEE*, 14(3) :54–62, 1999.

[153] M. Papazoglou. *Web services : principles and technology*. Addison-Wesley, 2008.

[154] M.P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing : State of the art and research challenges. *Computer*, 40(11) :38–45, 2007.

[155] F. Paternò, C. Mancini, and S. Meniconi. Concurtasktrees : A diagrammatic notation for specifying task models. In *Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction*, pages 362–369. Citeseer, 1997.

[156] C. Peltz. Web services orchestration and choreography. *Computer*, pages 46–52, 2003.

[157] S. Preuß. JESA Service Discovery Protocol : Efficient Service discovery in ad-hoc networks. *Lecture notes in computer science*, pages 1196–1201, 2003.

[158] Hausi A. Müller Mary Shaw Jesper Andersson Luciano Baresi Basil Becker Nelly Bencomo Yuriy Brun Bojan Cukic Ron Desmarais Schahram Dustdar Gregor Engels Kurt Geihs Karl M. Goeschka Alessandra Gorla Vincenzo Grassi Paola Inverardi Gabor Karsai Jeff Kramer Marin Litoiu Antonia Lopes Jeff Magee Sam Malek Serge Mankovskii Raffaela Mirandola John Mylopoulos Oscar Nierstrasz Mauro Pezza Christian Prehofer Wilhelm Schäfer Rick Schlichting Bradley Schmerl Dennis B. Smith Joa P. Sousa Gabriel Tamura Ladan Tahvildari Norha M. Villegas Thomas Vogel Danny Weyns Kenny Wong Jochen Wuttke Rogério de Lemos, Holger Giese. Software engineering for self-adaptive systems : A second research roadmap. Dagstuhl Seminar, May 2011.

[159] H. Schmeck, C. Müller-Schloer, E. Çakar, M. Mnif, and U. Richter. Adaptivity and self-organization in organic computing systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 5(3) :1–32, 2010.

Ce papier propose une approche permettant de faire émerger des systèmes en se basant sur les techniques d'organic computing. Un état de l'art définit bien les faiblesses des approches actuelles en terme de self-organisation. Le papier pose clairement la définition de l'émergence. .

[160] I. Sedov, S. Preuss, C. Cap, M. Haase, and D. Timmermann. Time and energy efficient service discovery in Bluetooth. In *Vehicular Technology Conference, 2003. VTC 2003-Spring. The 57th IEEE Semiannual*, volume 1, 2003.

[161] C. Szyperski, D. Gruntz, and S. Murer. *Component software : beyond object-oriented programming*. Addison-Wesley Professional, 2002.

[162] J. Whittle, P. Sawyer, N. Bencomo, B.H.C. Cheng, and J.M. Bruel. Relax : a language to address uncertainty in self-adaptive systems requirement. *Requirements Engineering*, 15(2) :177–196, 2010.

Ce papier présente Relax, un langage pour définir des *requirements* pour des systèmes auto-adaptatifs. Ce langage permet d'exprimer une certaine imprévisibilité. Deux types sont identifiés : l'imprévisibilité environnementale (des requirements qui restent valident malgré des changements non-anticipés de contexte) et l'imprévisibilité comportementale (changements dans les requirements qui n'avaient pas été prévus). Le langage proposé est pseudo-naturel et sa sémantique est exprimée en terme de logique floue. .

[163] F. Zhu, M.W. Mutka, and L.M. Ni. Service discovery in pervasive computing environments. *IEEE Pervasive Computing*, 4(4) :81–90, 2005.

Table des figures

Liste des tableaux